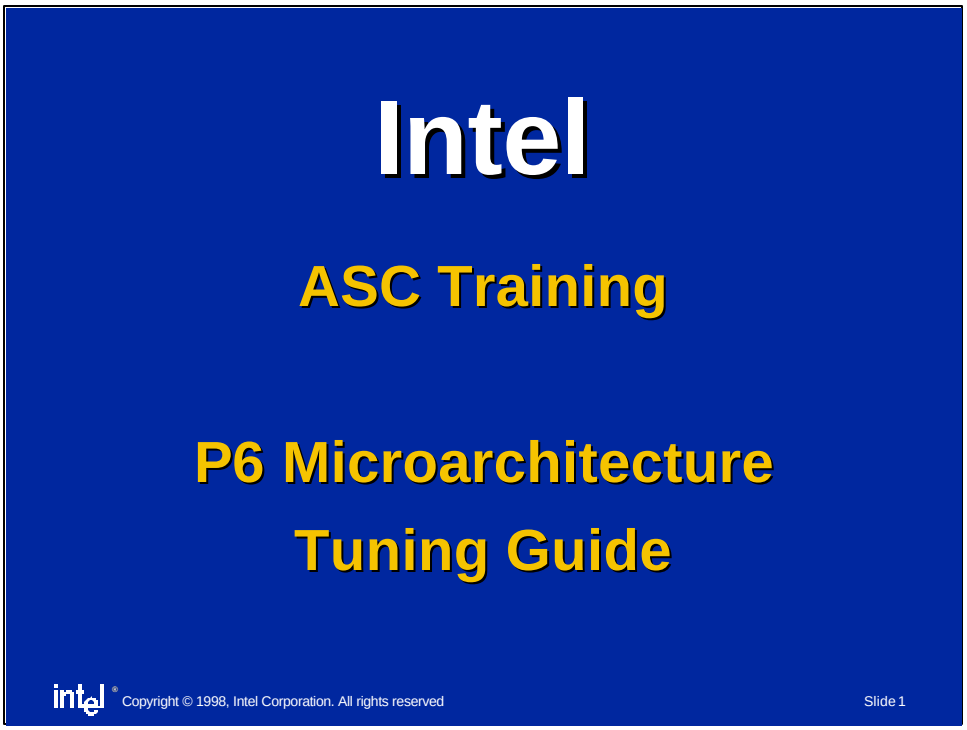
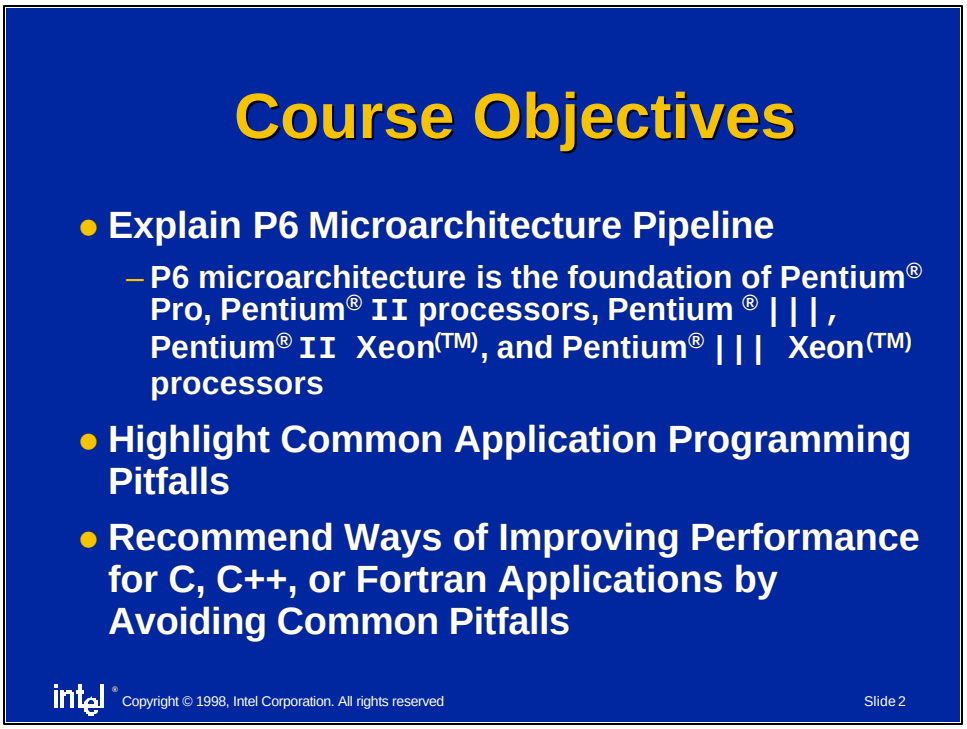


Notes

This talk is concerned with processor microarchitecture level performance tuning for applications written to run on Intel IA-32 architecture platforms. These slides provide detailed information on how applications can take advantage of the high performance capabilities of Intel P6 microarchitecture processors and platforms.

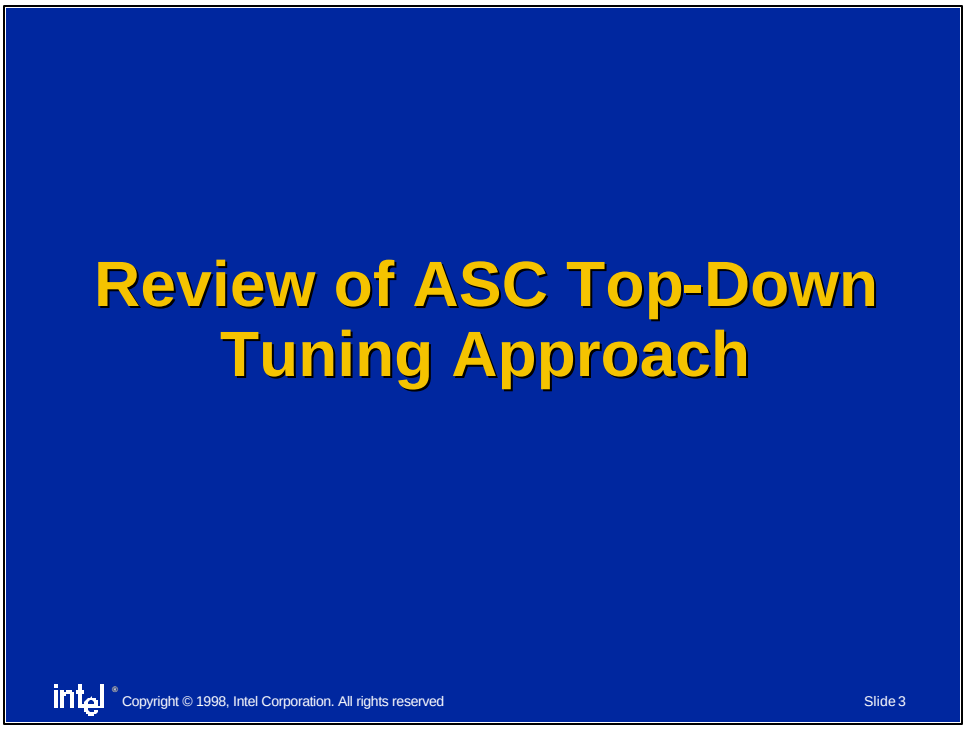
It is assumed that the audience of this talk is familiar with performance tuning terminology and concepts, and have done some high level and assembly language programming.



Notes

The discussion starts with a review of the P6 microarchitecture design and its implications for application performance tuning. Detailed description of the P6 microarchitecture - the foundation of the Pentium(r) Pro processor, Pentium(r) II processor, Pentium(r) II Xeon™ processor, Pentium(r) III and Pentium(r) III Xeon™ - is given. Each stage of the microarchitecture pipeline is discussed; methods for exploiting each stage for optimal application performance is exposed.

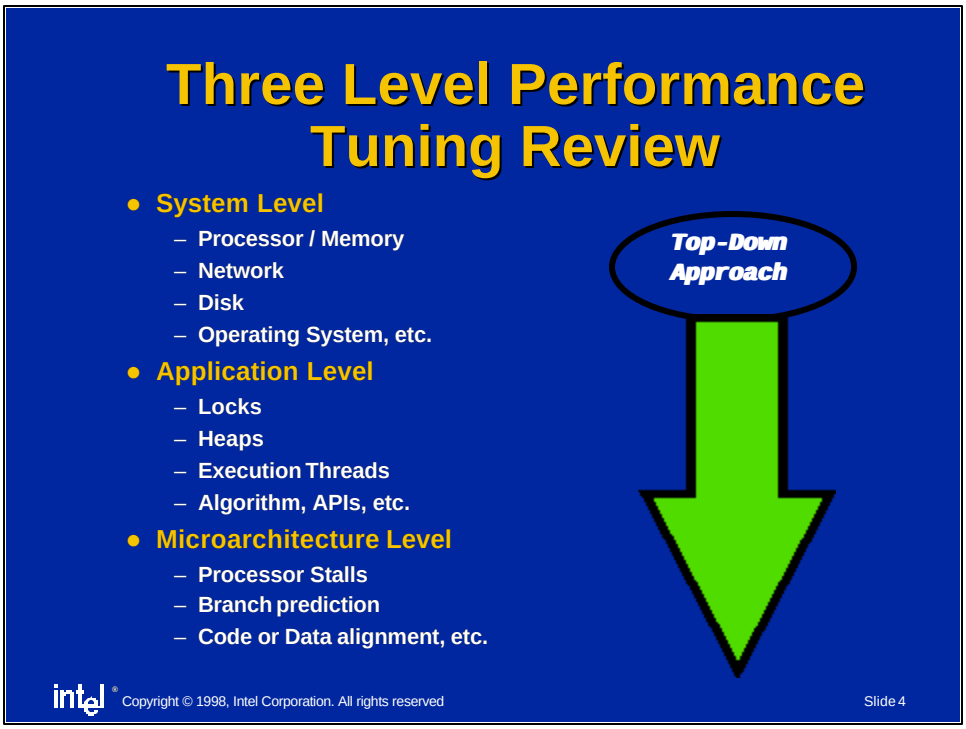
Common pitfalls that are encountered in the design and implementation of applications for the P6 microarchitecture processors and platforms are listed. Various methods for avoiding the pitfalls are also discussed in details. Many examples on how C and IA-32 assembly language programs can be implemented to avoid the most common pitfalls are given.



Notes

Within Intel ASC lab, the microarchitecture level tuning is viewed as one part of a multi-level tuning methodology.

This section is a review of the ASC top-down performance tuning approach.



Notes

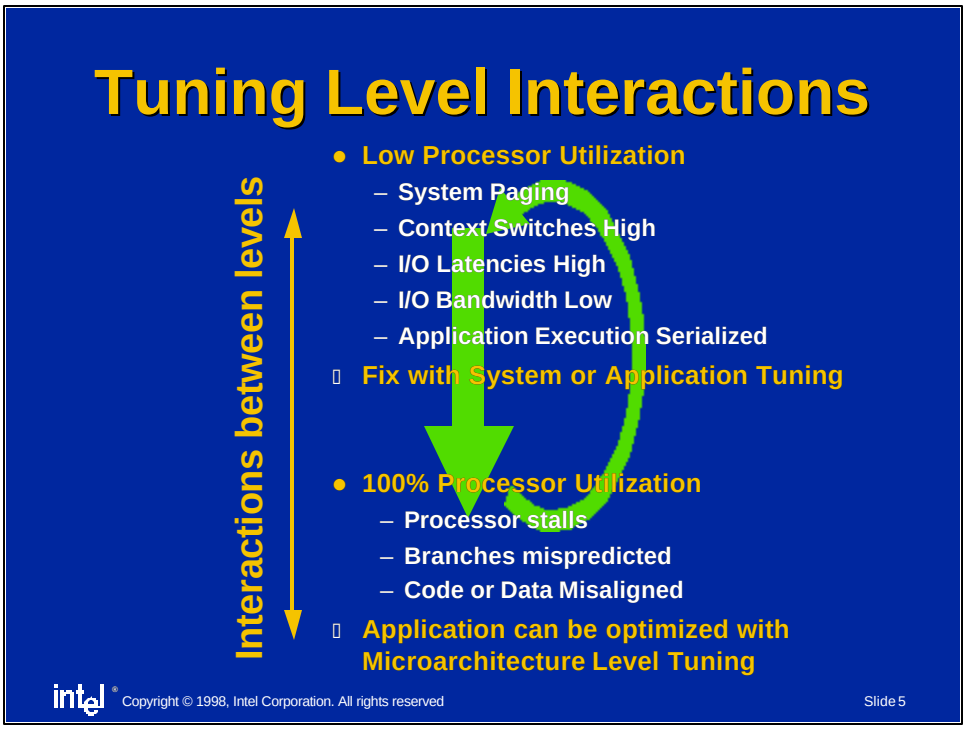
Intel ASC methodology emphasizes a three level approach to performance tuning - System Level, Application Level, and Microarchitecture Level performance tuning.

System level tuning involves making changes to the Operating System and the hardware platform. At the system level, processors, memory, disk, and network devices are added as needed for optimal performance and price/performance of applications. Devices are also tuned and configured to meet the demand of applications running on the system.

Application level tuning involves making changes to an application to eliminate bottlenecks and inefficiencies inherent in the application code. Locks are implement in ways that minimizes their serialization of application execution. Smarter heap allocations and de-allocations are implemented to minimize overheads. Better Application Program Interface (API) calls are chosen to minimize application serialization and API call overheads. Also, opportunities for multi-threading of applications should be explored at this level.

Microarchitecture level tuning involves implementation of applications in ways that allow them to take full advantage of processor hardware. Applications are written to avoid events that cause the processor to block or become inefficient.

These three levels form the cornerstone of an iterative tuning methodology. A top-down approach to the three level tuning is emphasized. The methodology requires that the System level tuning is done first followed by the Application level tuning and finally the microarchitecture level tuning. The work at each level continues until no performance gain can be achieved. At the end of the microarchitecture level, the process start again from the System level.



Notes

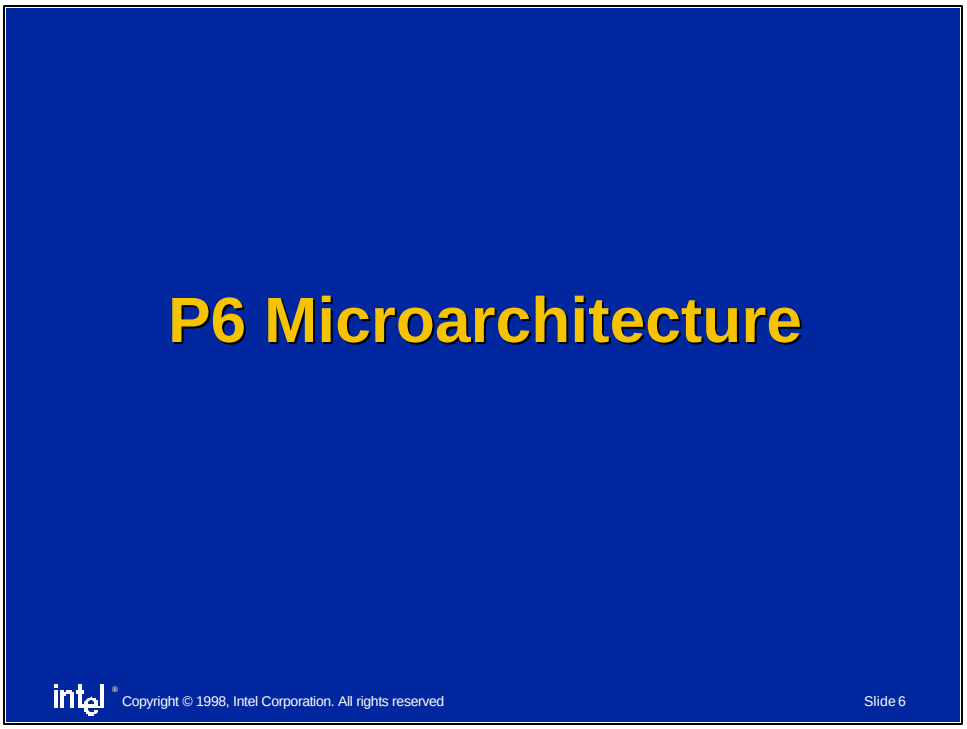
Performance issues can be in one of two states as far as the processor is concerned - either the processor is the bottleneck, or the processor is not the bottleneck in the system.

The processor cannot be the bottleneck in a system if the system has a CPU utilization less than 100% (or very close to 100%). A system without 100% processor utilization has bottlenecks elsewhere; the bottleneck could be in the I/O subsystem, the Operating System, or the application. The performance of applications that exhibit less than 100% processor utilization can be improved with system and application level tuning. Minimal or no performance gains can be expected from microarchitecture level tuning for such applications.

The processor is the bottleneck for an application when the application has a processor utilization of 100%. Such an application may benefit from microarchitecture tuning that results in a more efficient execution of the application instruction stream.

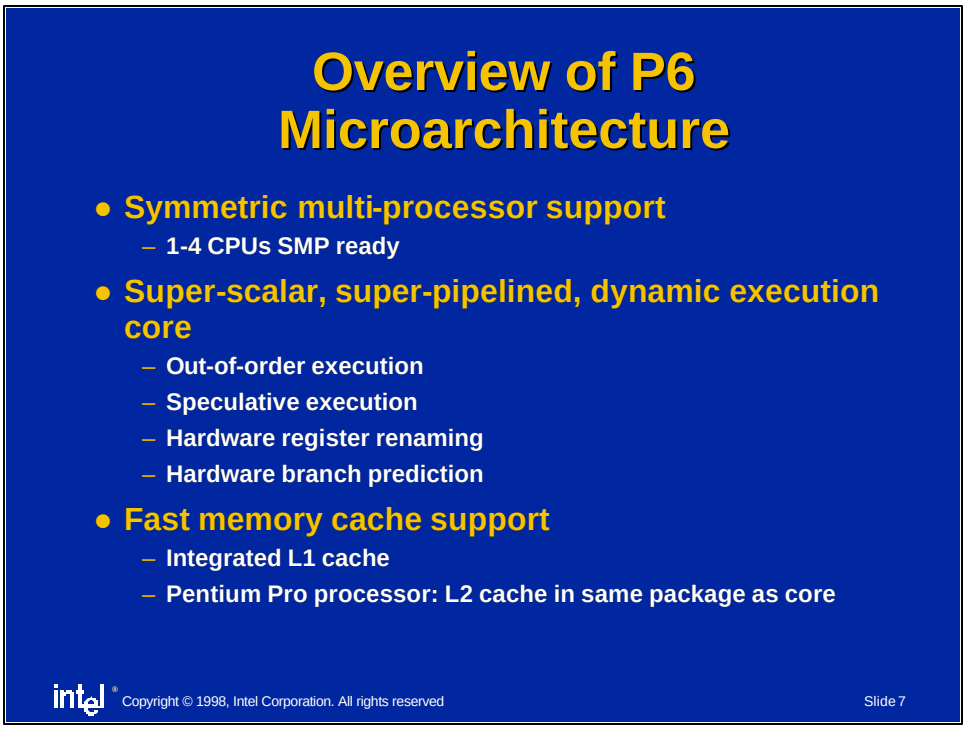
An application may have 100% CPU utilization because it is executing too many instructions per operation. The performance of such an application may be remedied by re-writing the application to use better algorithm and API calls, and incur less Operating System overheads.

It is possible to continue microarchitecture level tuning of an application until the processor is no longer the bottleneck. At such point, it is prudent to move the tuning effort to System and Application level tuning.



Notes

This section starts the discussion on the design of P6 microarchitecture.



Notes

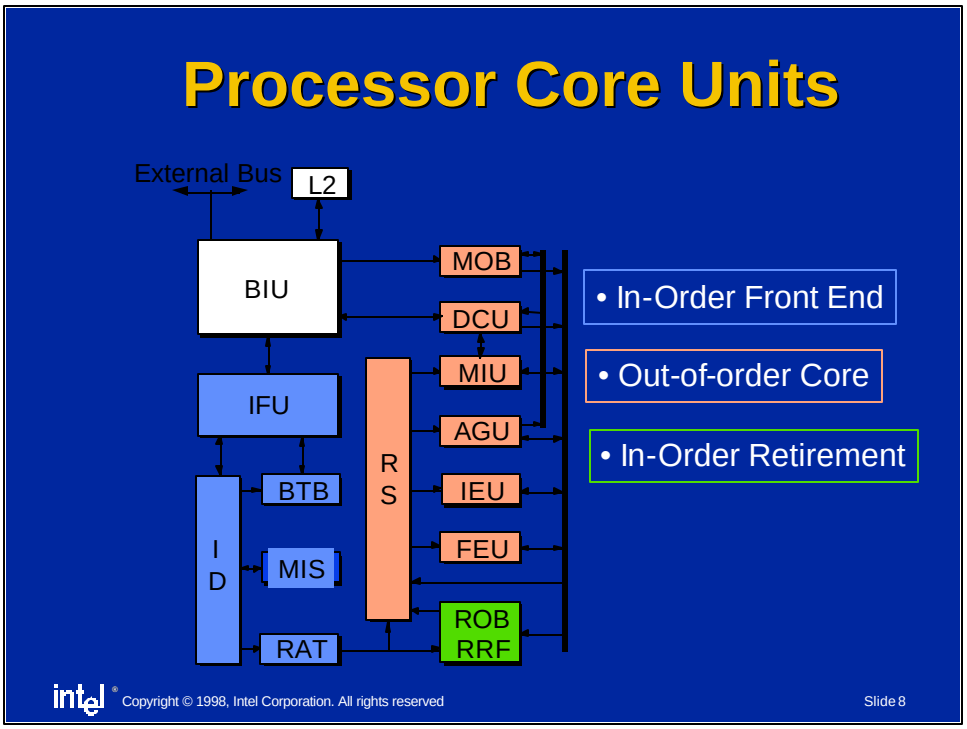
The P6 microarchitecture combines the benefits of a Complex Instruction Set Computer (CISC) with the benefits of a Reduced Instruction Set Computer (RISC).

The microarchitecture introduces several performance enhancements to IA-32 applications. It provides the benefits of a new design without requiring old IA-32 applications to be ported to a new architecture.

The P6 microarchitecture processors are super-scalar because they can execute more than one instruction per cycle. They are super-pipelined because they have many more stages than other comparable processors. The P6 microarchitecture processors support dynamic execution through speculative and out-of-order execution.

Among the new enhancements in the P6 microarchitecture are hardware register renaming, speculative execution, branch prediction and out-of-order execution. Hardware register renaming allows the number of processor registers to be increased without requiring IA-32 applications to be re-written to take advantage of the additional registers. Speculative execution means that instructions are executed before all conditions before them are known. Branch prediction allows for a more efficient utilization of the processor pipeline. Out-of-order execution allows instructions to be executed in any order that make sense for the processor.

The P6 microarchitecture supports two levels of fast memory cache - the L1 and L2 cache. The details of the microarchitecture is discussed in the following slides.

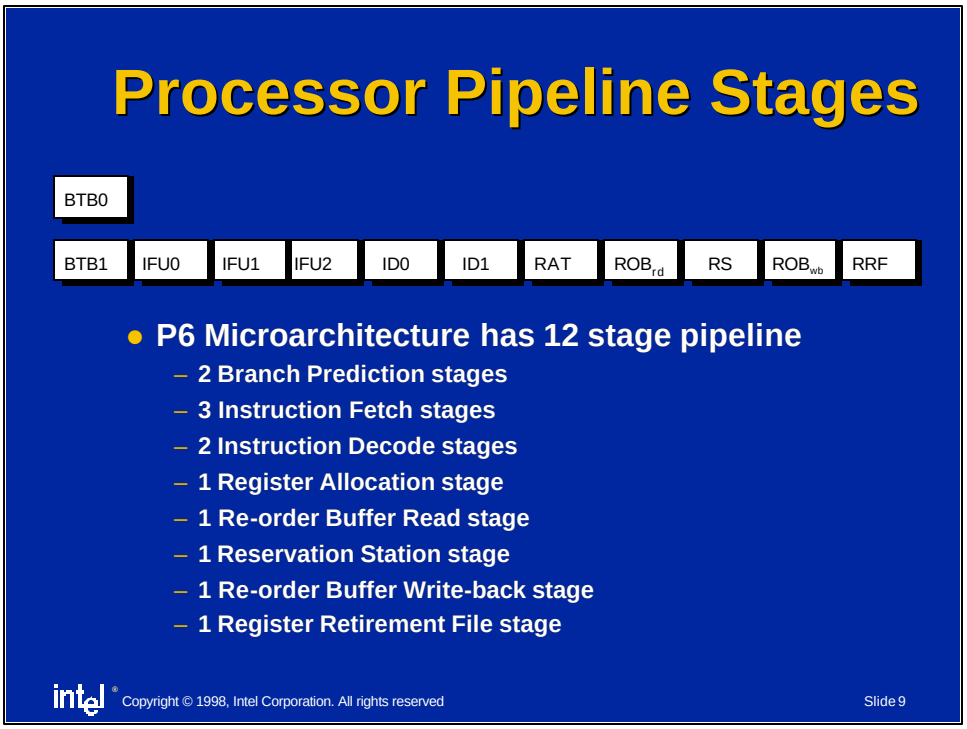


Notes

The P6 microarchitecture is made up of in-order front end, out-of-order core and in-order retirement units.

The front end includes Instruction Fetch, Instruction Decode, Branch Target Buffer, Micro-instruction Sequencer, and Register Address Table units. The out-of-order core is made up several execution units; the units include Floating Point Execution units, Integer Execution units, and Address Generation units. The in-order retirement back end includes the Re-order Buffer and the Register Retirement File units.

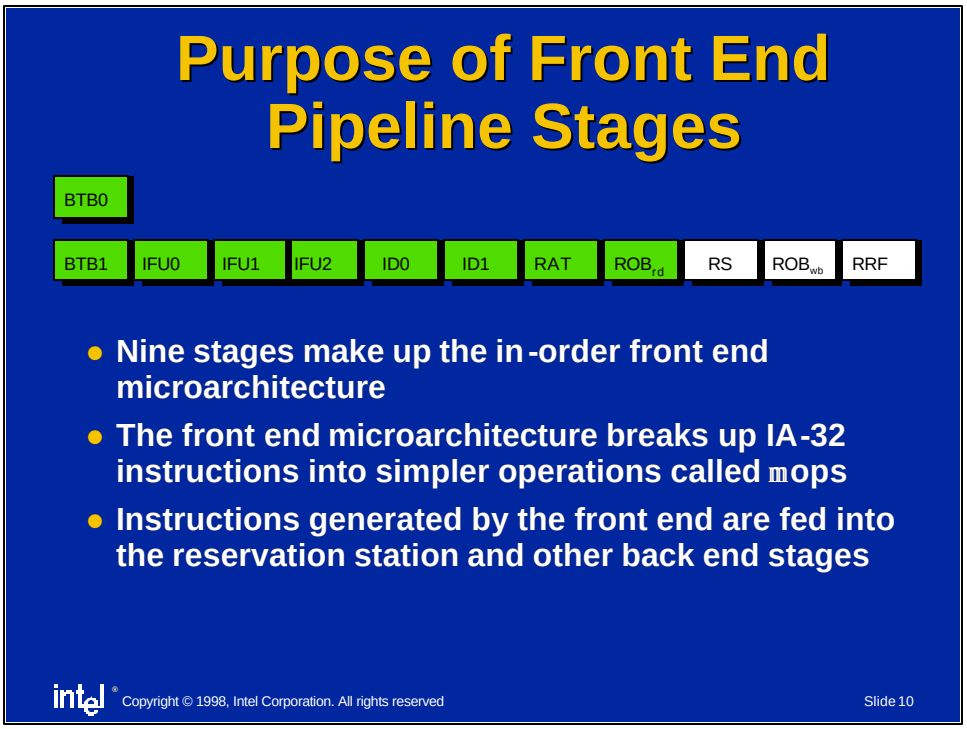
The following slides illustrate the steps that an instruction take inside a P6 microarchitecture processor.



Notes

The P6 microarchitecture has 12 pipeline stages that an instruction would take to complete. Each pipeline stage is designed to prepare the instruction for a proceeding stage; the stages are taken in sequence until an instruction is completed and its results written to a register or memory.

The first five stages are concerned with predicting branches, and fetching instructions from memory. The next four stages decode instructions and prepare them to be executed in parallel and out of order by the super-scalar execution engine. One stage executes instructions. The final two stages prepare and write values back to registers and memory.

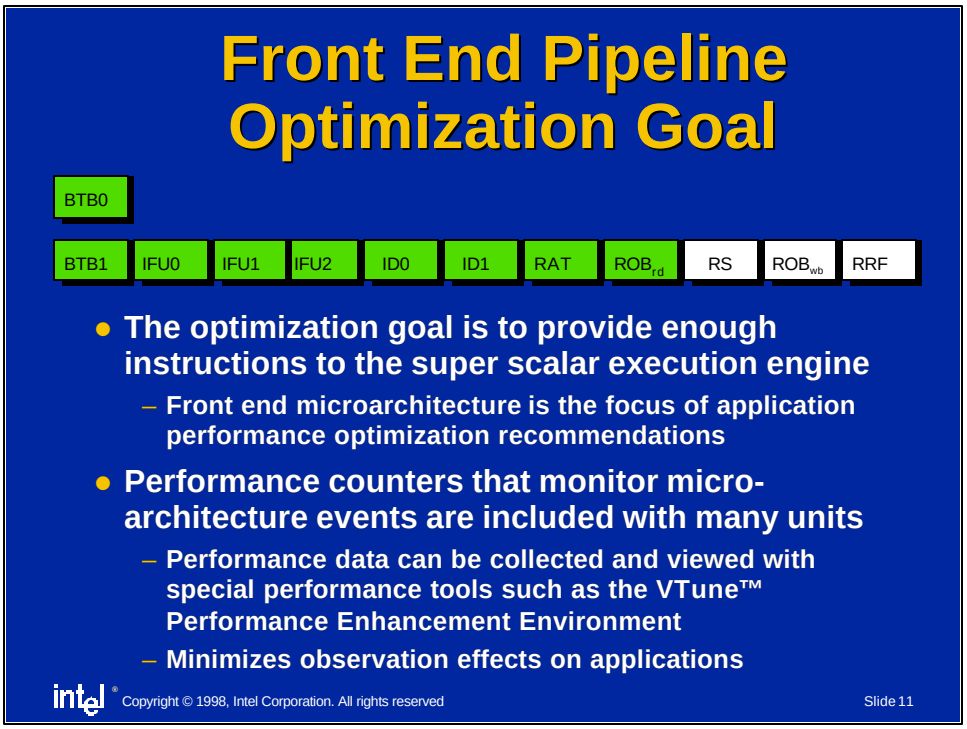


Notes

Application performance tuning recommendations for the P6 microarchitecture are focused on the first nine pipeline stages - the front end microarchitecture stages.

These nine front end stages break up IA-32 instructions generated by compilers and assemblers into simpler micro-operations called µops. These µops are executed by the super-scalar execution engine. Results of instruction executions are passed on to the back end to be written back to registers or memory.

The way applications are written impact the performance of the front end microarchitecture the most. Applications have no direct control of how the execution engine and the back end of the microarchitecture work. For the most part, the execution engine and back end would do the right thing given optimal performance of the front end.



Notes

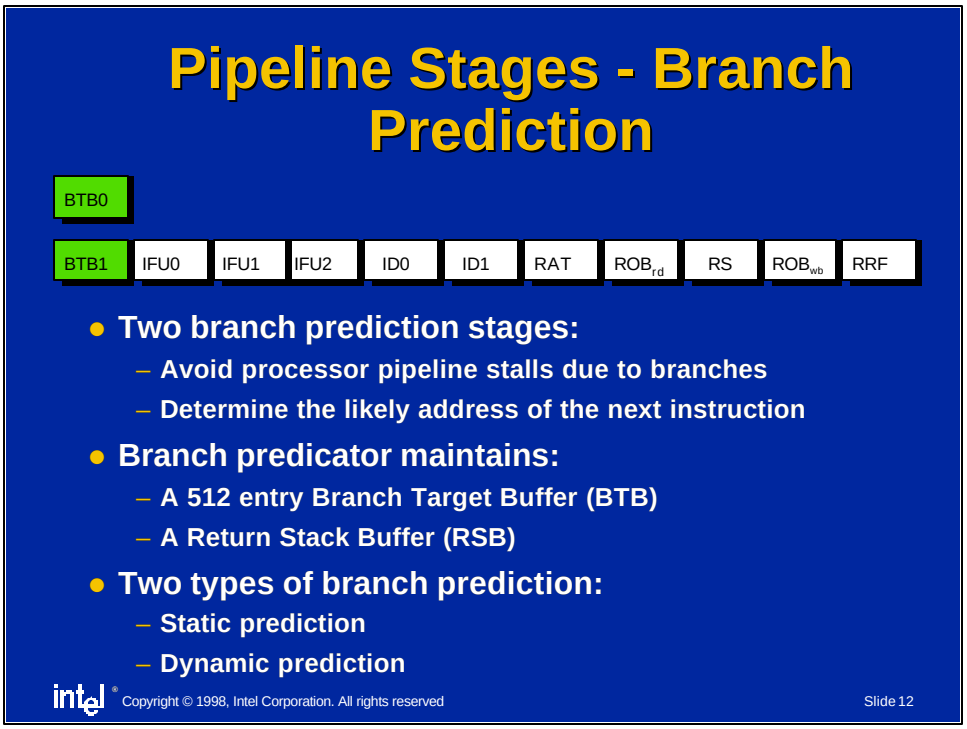
By increasing the performance of the front end microarchitecture for an application, overall processor performance of the application is also increased.

When the throughput of the front end microarchitecture is increased, enough instructions are available for the execution engine to keep each execution unit busy at each CPU cycle. This in turn would likely increase the throughput of the back end: hence, overall microarchitecture performance.

There are many performance counters included to monitor events on various P6 microarchitecture units. Some of the events can be used to monitor the performance of the pipeline.

Because the P6 microarchitecture has an out of order execution engine, the dynamic flow of instructions in an application is important to actual performance of the application. Applications need to be monitored in ways that maintain the correct order of instructions. Using tools that instrument applications (ie. by adding instructions that collect various performance statistics) will likely perturb the dynamic behavior of applications on the processor. Hence, monitoring processor performance by application instrumentation is not the most reliable way of monitoring the performance of P6 microarchitecture processors.

Circuits were added to the P6 microarchitecture to asynchronously count microarchitecture events as they occur in the processor pipeline. This allows for collection of performance data without disturbing the order of instructions. The microarchitecture event counters and performance data can be viewed with minimal overhead using special performance tools such as the VTune™ Performance Enhancement Environment.

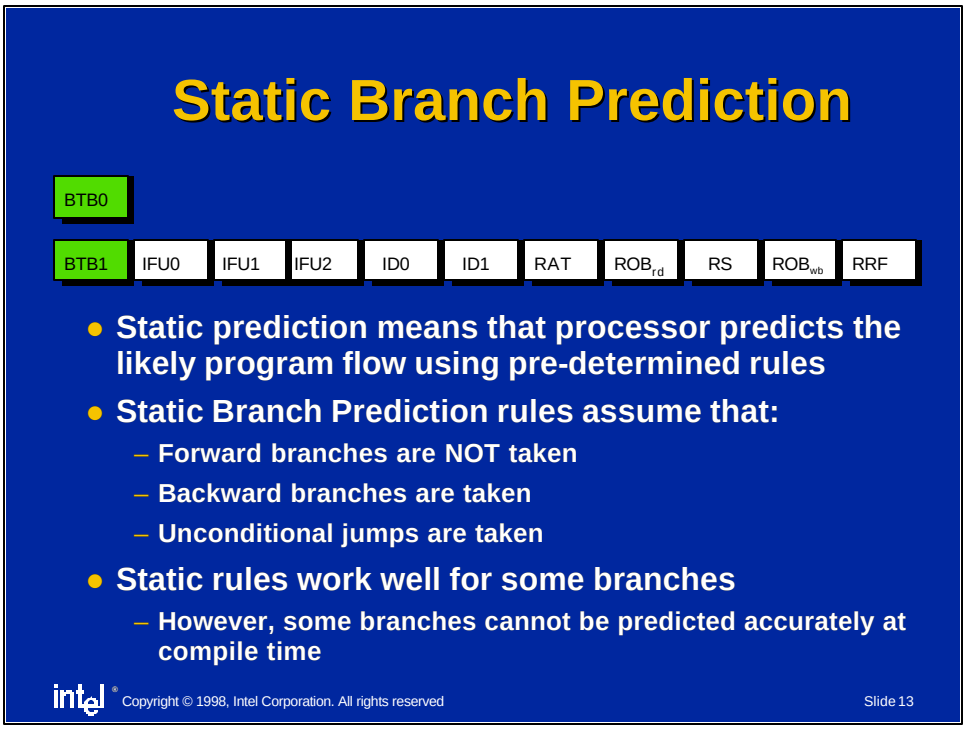


Notes

The first two stages of the P6 microarchitecture pipeline is used to predict branches. Branch prediction is necessary to avoid processor stalls due to branches. Pipelined processors need to predict branches in order to keep each pipeline stage busy with instruction. Because there are more pipeline stages in a super-pipelined microarchitecture, branch prediction is extremely important.

Branch prediction within a processor hardware means that the processor predicts whether an instruction would cause the execution of an application to be transferred to a new address (i.e. a new location other than the next linear address). The P6 microarchitecture reserves a 512 entry Branch Target Buffer (BTB) and a Return Stack Buffer which it uses to predict branches.

The microarchitecture supports two forms of prediction - static and dynamic branch prediction. Both methods are very useful for predicting the behavior of branches at runtime.



Notes

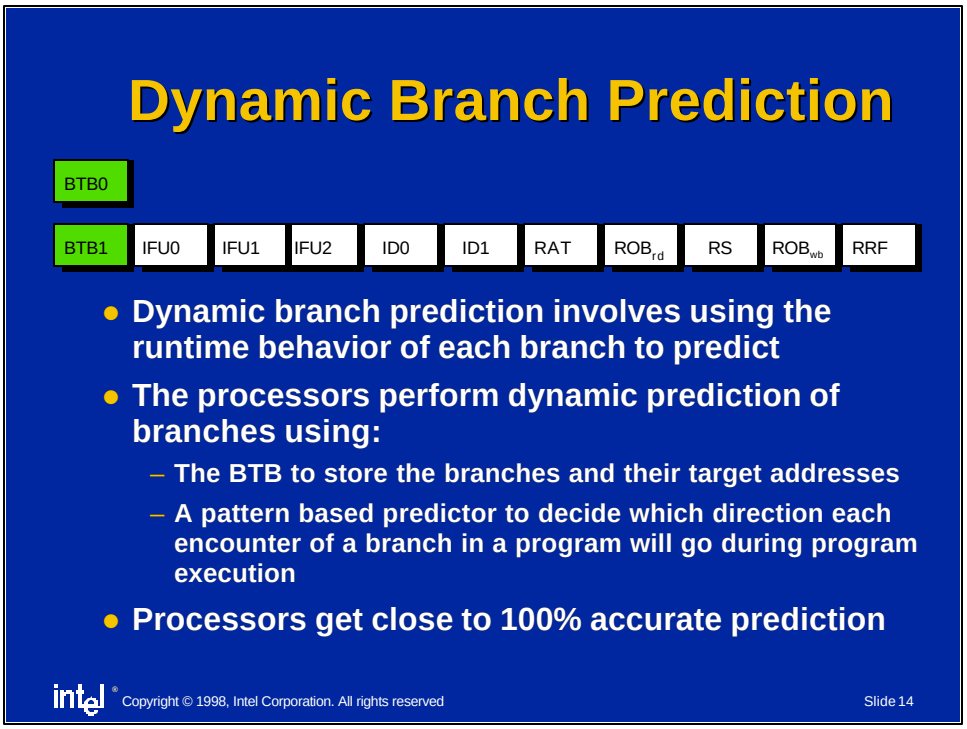
Processors can predict branches based on a static set of rules. A compiler (or programmer) can generate a sequence of instructions for an application according to what is known about each branch at compile (or development) time. A processor could make a fairly accurate predictions on the behavior of some branch instructions based on the sequence of instructions generated by a compiler.

The P6 microarchitecture supports this kind of static branch prediction. The microarchitecture performs static branch prediction using the following rules:

- Branches to addresses greater than the current Instruction Pointer(IP) are assumed (and predicted) not taken
- Branches to addresses less than the current IP are predicted taken
- Hard jumps (I.e. unconditional branches, calls, and returns) are predicted taken

Based on these rules, a compiler can generate a sequence of instructions at compile time that make the processor's runtime static prediction accurate.

Even though static predictions work well for certain branches, information on how a branch will behave at runtime may not be available at compile (or development) time. Since some branch instructions may be dependent on variables available only at runtime, the behavior of some branches may be available only at runtime. Also, since some branch instructions may depend on the outcome of previous branches, there may be a cascading behavior of branches at runtime. Therefore, it may not be enough for a super-pipelined processor to support only static branch prediction.

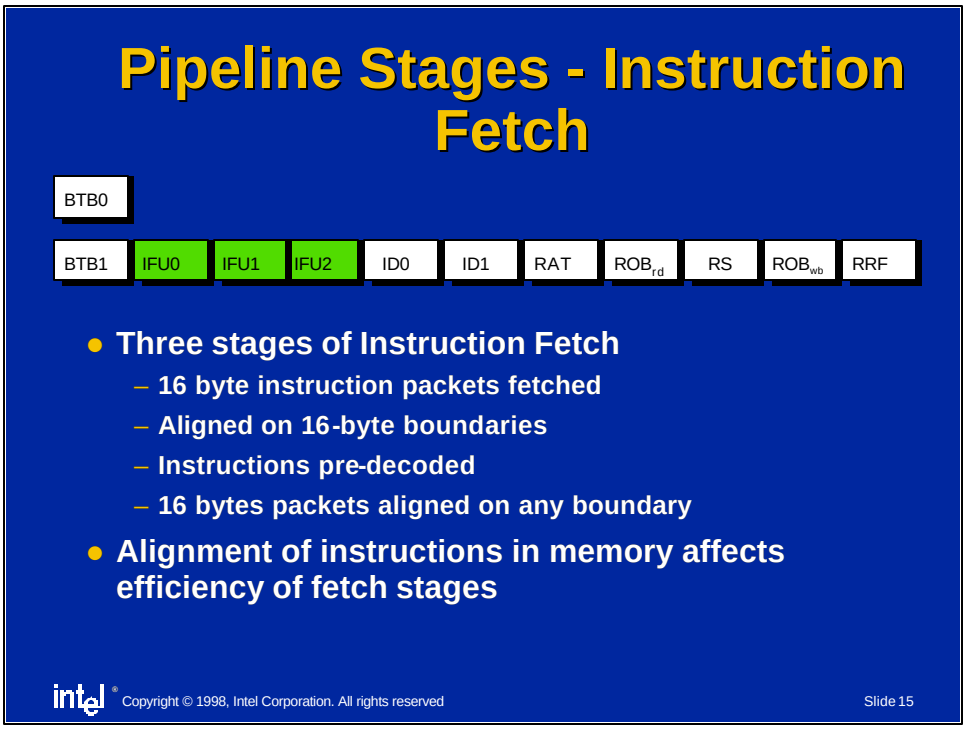


Notes

Hence, in addition to static branch prediction, the P6 microarchitecture processors perform branch prediction based on runtime (i.e. dynamic) behavior of branch instructions.

The processors perform dynamic branch prediction using history information about branch instructions. The processors store each branch, history and target address in a 512 entry BTB. Using the information in the BTB, the processors dynamically predict branches and their target addresses at runtime.

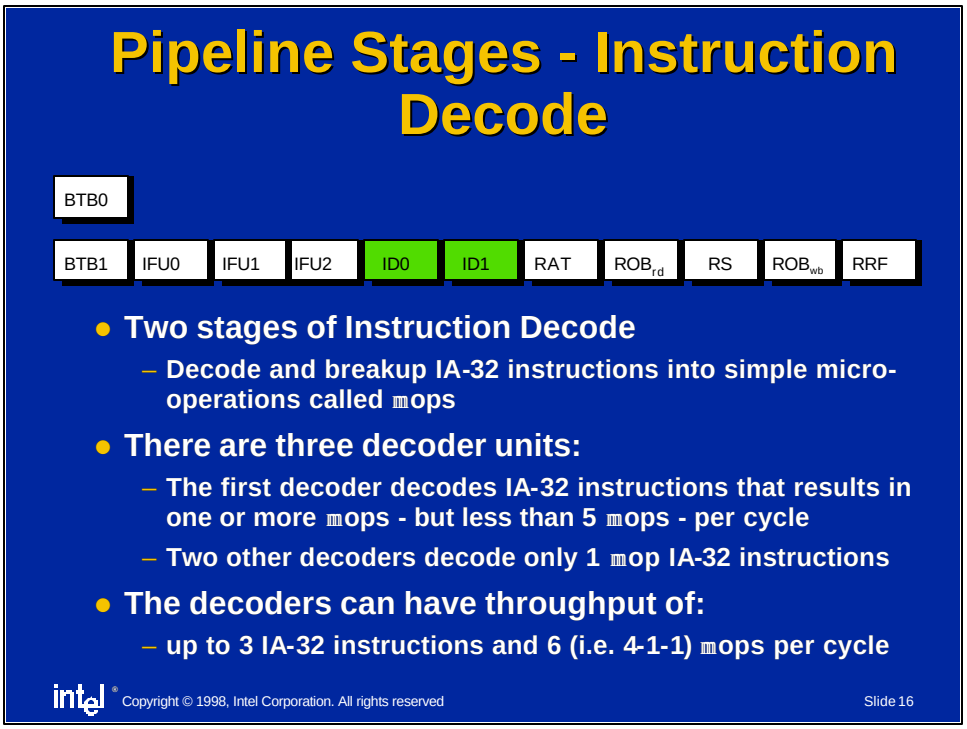
The combination of static and dynamic branch predictions results in a very accurate prediction rate for well written applications.



Notes

After two stages of branch prediction, an instruction must go through three stages of instruction fetch. During these three stages, 16 bytes of instructions are fetched, pre-decoded and aligned for the decode stage.

The alignment of instructions in memory could have significant performance impact for the fetch stages. Application optimization goals include alignment of instructions in memory in ways that increase efficient utilization of all 16 bytes of instructions fetched at each cycle. More information on alignment is provided later in this presentation.

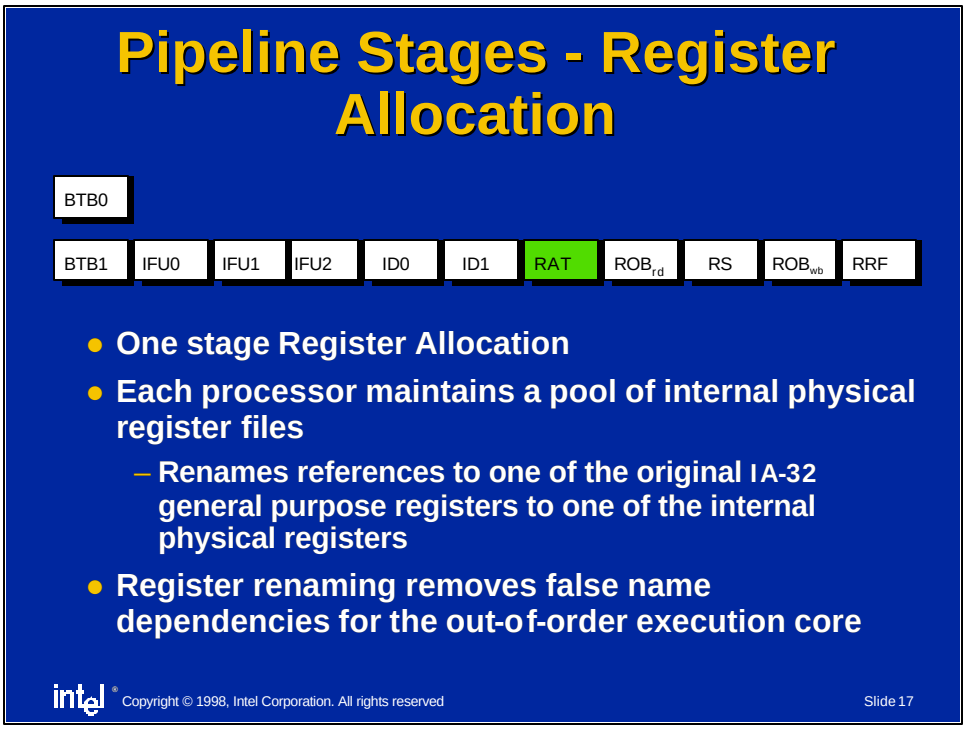


Notes

Instructions go through two stages of instruction decode. During these two stages, IA-32 instructions are broken up into micro-operations called µops.

The microarchitecture has three decoders that work in parallel. The first decoder decodes complex and simple(I.e 1 µop) IA-32 instructions while the last two decoders decode only simple instructions. The first decoder decodes instructions that generate 1 to 4 µops in one cycle. Instructions that generate more than 4 µops take more than one cycle to decode.

The optimization goal for the decode stages is to generate a sequence of instructions that can be decoded in parallel by the three decoders. This is usually the 4-1-1 sequence: that is a sequence of complex followed by two simple instructions.

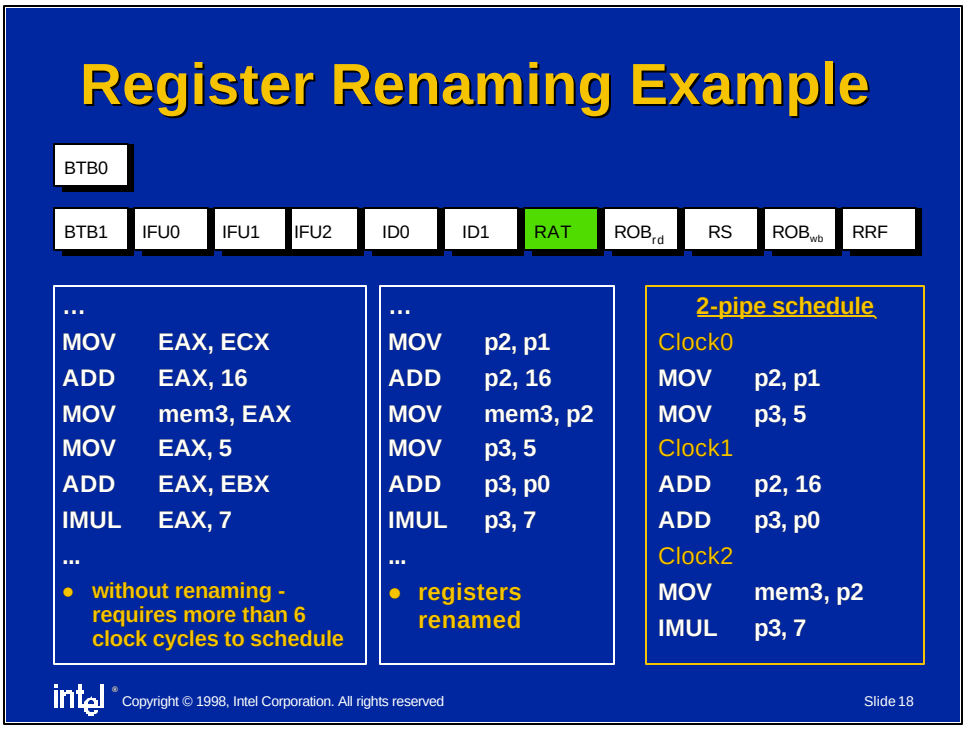


Notes

The P6 microarchitecture maintains a pool of internal registers. The number of internal registers are much greater than the programmer visible set of 8 registers in IA-32 architecture.

After decode, instructions go through one stage of Register Allocation Table (RAT). During the RAT stage, IA-32 register references by an instruction are renamed to references to registers in the internal register set.

Register renaming removes false register name dependencies between instructions. By removing false register name dependencies, the microarchitecture uncovers truly independent instructions that it can execute in parallel. Large number of independent instructions helps to keep the execution units busy and improves overall throughput of a P6 microarchitecture based processor.

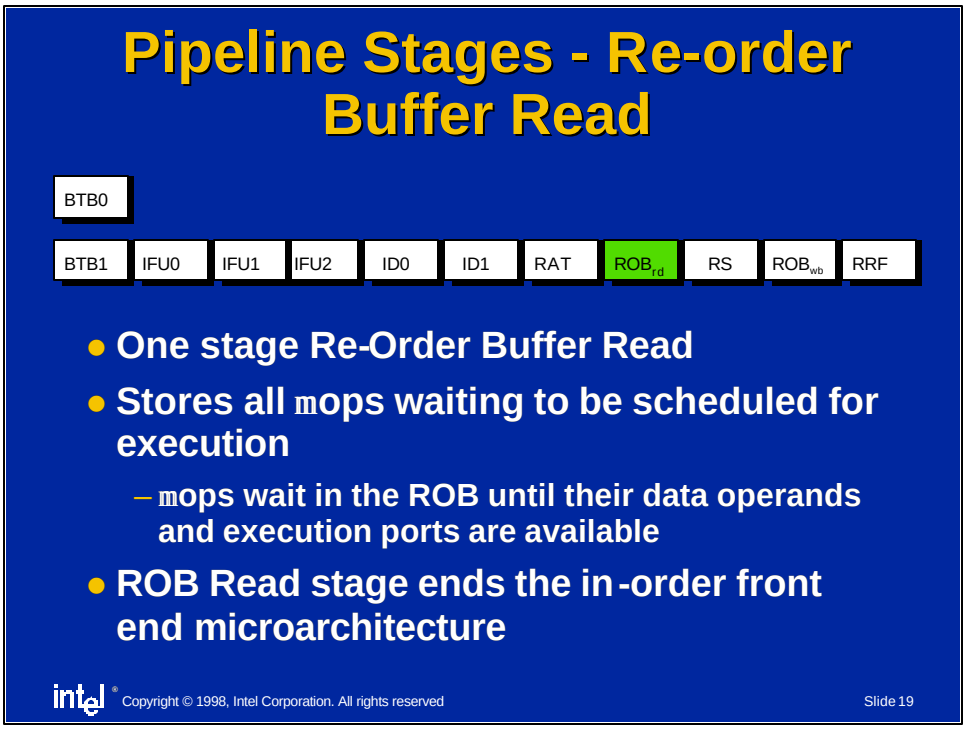


Notes

The following example illustrates how Register Renaming works.

The microarchitecture renames all references to EAX, EBX, and ECX to internal register names p0, p1, p2, and p3. The names p0, p1, p2, and so on - used in the example here - are made up; the actual names of the internal registers are not published and cannot be accesses by the programmer or compiler.

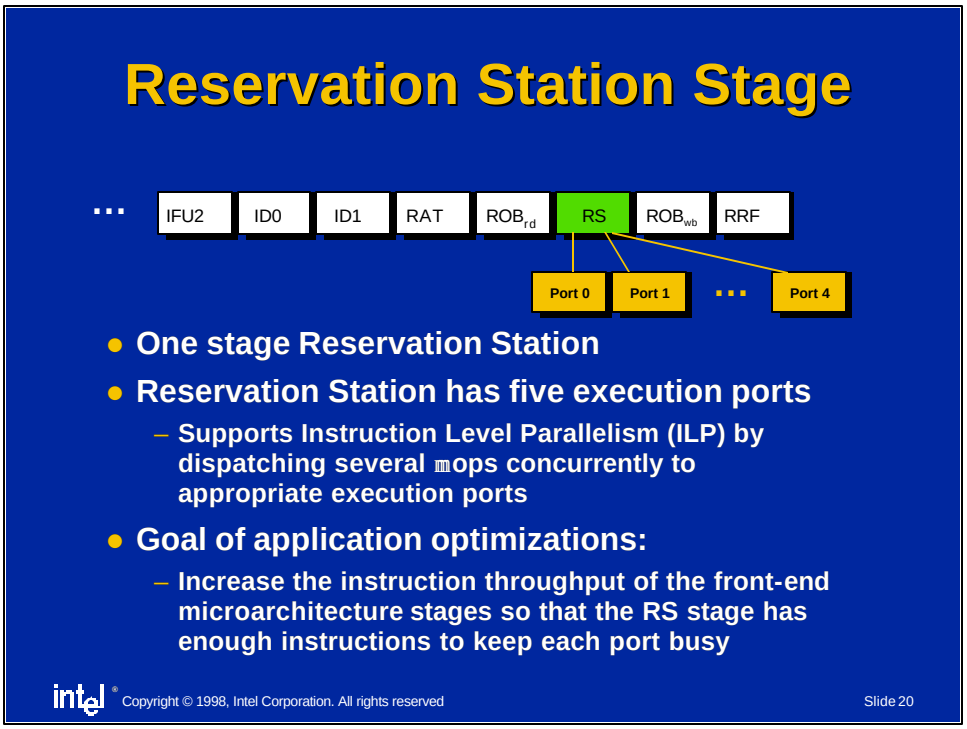
In the above example, two instances of EAX are identified. A new internal register is assigned each time a new instance of a register reference is seen by the processor hardware. After the register renaming, the example shows how two instructions can be scheduled for parallel execution at each cycle.



Notes

After register renaming, instructions are inserted into a Re-Order Buffer(ROB) during the one stage ROB_{rd}. The ROB_{rd} stage is the end of the in-order front end microarchitecture.

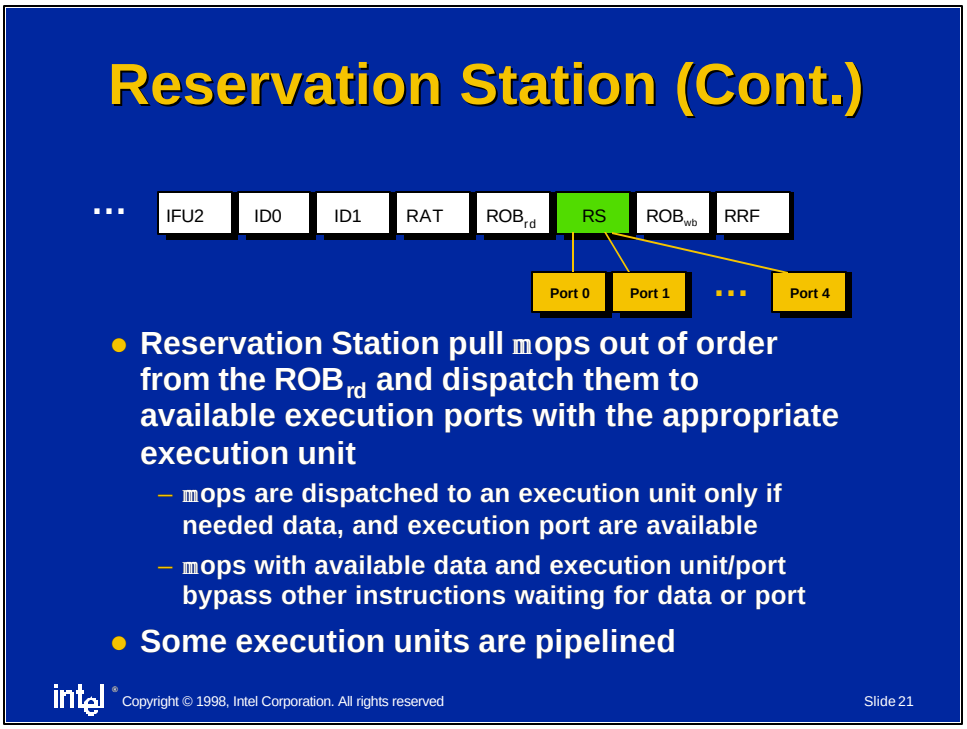
Instructions wait in the ROB until they can be scheduled for execution. Instruction can be scheduled for execution only after all data dependencies are resolved and there are execution ports where they can be scheduled.



Notes

Instructions are executed at the Reservation Station (RS) stage - after all data dependencies have been resolved. The Reservation Station maintains five execution ports to facilitate instruction level parallelism (ILP); up to five instructions can start execution at a cycle.

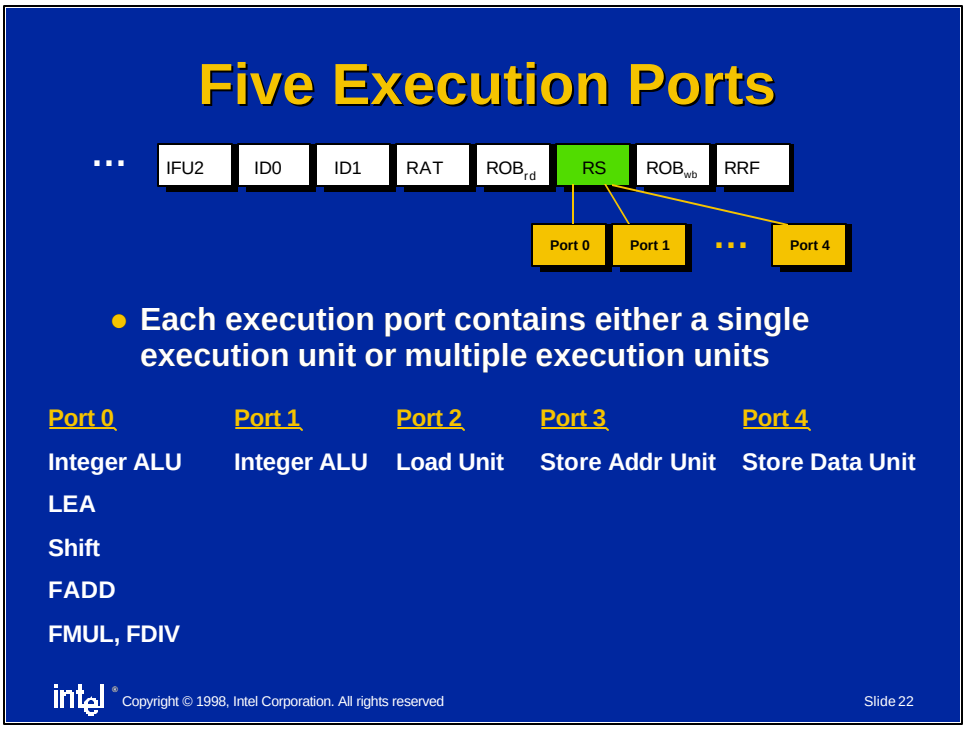
This stage is the motivation for all the application tuning suggestions made to optimize the throughput of the front end microarchitecture for each application. If the throughput of the front end microarchitecture is high, there will be a mix of independent instructions in the ROB that can be scheduled in parallel. The probability that every execution port remains busy at every cycle is increased as large number of instructions become available for the Reservation Station to dispatch.



Notes

Instructions can be scheduled out-of-order from the ROB. While an instructions is waiting for data from memory (or previous instructions that it has dependencies with), proceeding instructions can be scheduled for execution. Out-of-order execution maximizes the throughput and utilization of the execution units.

Instructions are executed speculatively when all control dependencies (such as branches) may not have been resolved. Speculative execution do not result in incorrect execution since no changes made by an instruction execution is visible until the instruction is retired. Mispredicted branches are detected and recovered during retirement.



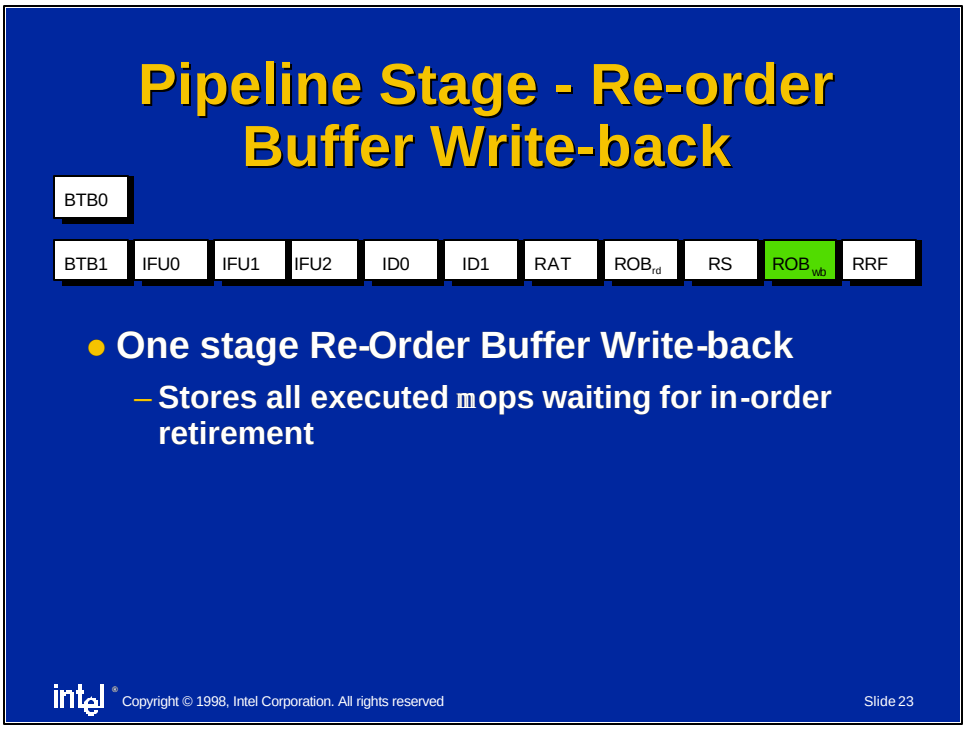
Notes

The five execution ports have different execution units attached to them. Attached to the first port (Port0) are integer ALU, Load Effective Address, Shift and Floating Point execution units. Port1 has only an Integer ALU execution unit attached. Ports 2, 3 and 4 has Load, Store Address and Store Data execution units respectively.

The latency of the execution units vary significantly. Some of the execution units take only one cycle to complete instructions while other units take more than one cycle per instruction. However, because some of the the high latency execution units are pipelined, instructions can be scheduled to complete with a throughput of one instruction per CPU cycle on many of the execution units.

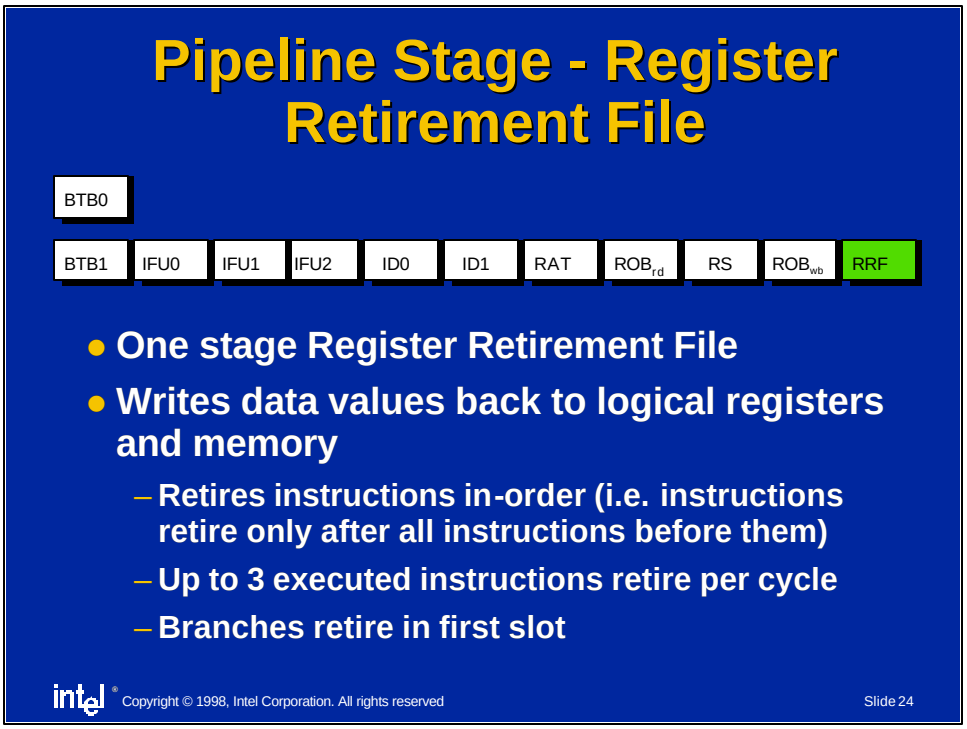
The Integer ALU execution units complete instructions with a latency of one cycle and a throughput of one instruction per cycle. The floating point add (FADD) execution unit completes instructions with latency of three cycles. However, FADD is pipelined; instructions can be scheduled on the FADD unit at every cycle for a throughput of one instruction per cycle. The floating point divide (FDIV) execution unit takes 17 CPU cycles for single precision division and 36 cycles for double precision division. The FDIV unit is not pipelined.

The Intel Architecture Optimization Manual has the complete list of execution unit latency and throughput.



Notes

Executed instructions get inserted into the ROB during the ROB_{wb} stage. After the RS stage, instructions stay in the ROB until all preceding instructions have been retired. The ROB_{wb} stage is the beginning of the in-order back end microarchitecture.

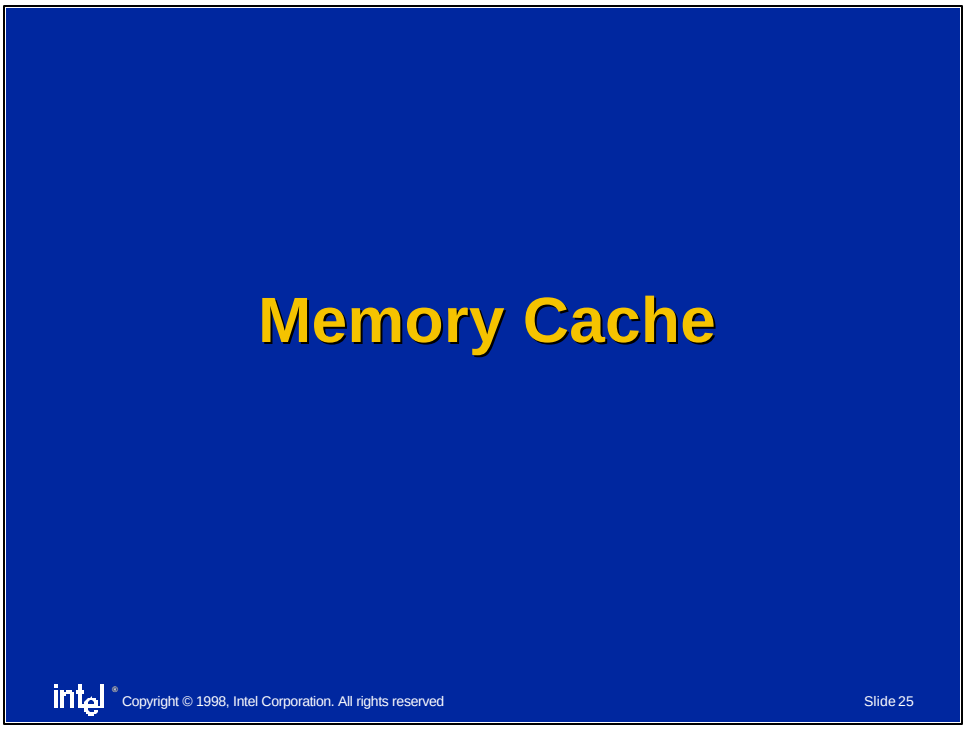


Notes

The last stage of the P6 microarchitecture pipeline is the Register Retirement File(RRF) stage. During the RRF stage, the values produced by instructions are written back to memory or actual IA-32 registers that were referred to before Register Renaming.

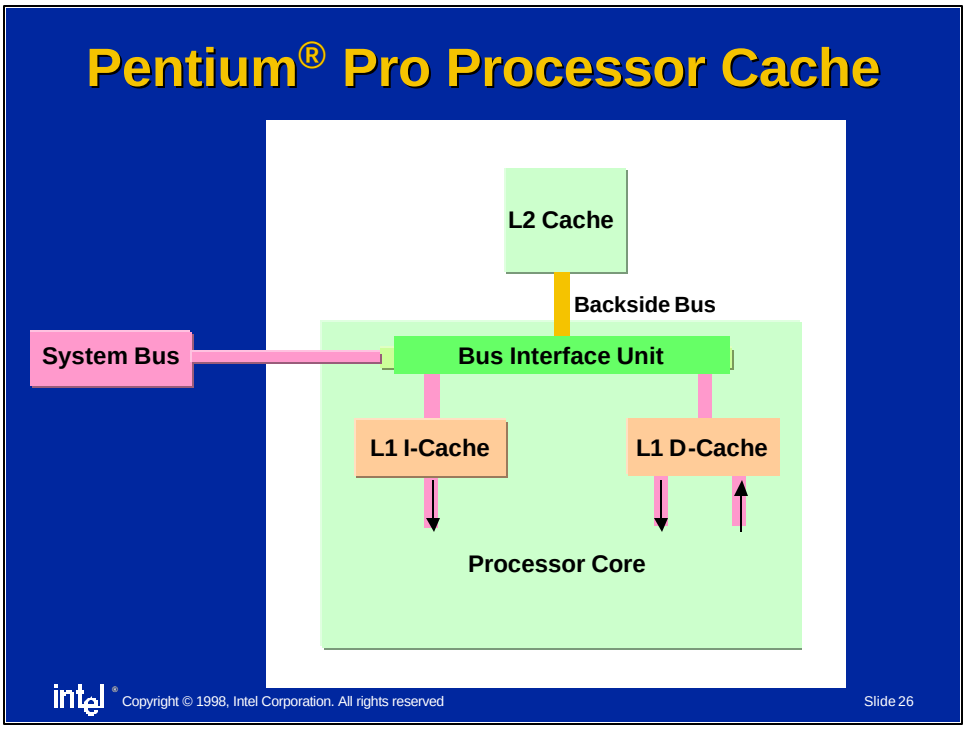
To support the ‘precise exception’ implemented by the IA-32 architecture, any exceptions generated during instruction execution in a P6 microarchitecture processor is visible only during the RRF stage.

Instructions retire only after all other instructions before them has been retired. Up to three instructions are retired per cycle. Branch instructions must retire in the first of the three retirement slots.



Notes

For over twenty years, processor speed has been doubling every 18 months. Similar changes have not occurred with DRAM technology. For the foreseeable future, it appears that the gap between processor and memory speed will continue to widen. Level one and level two caches are attempts to minimize the effects of memory on the processor by having a relatively fast memory between main memory and the processor.

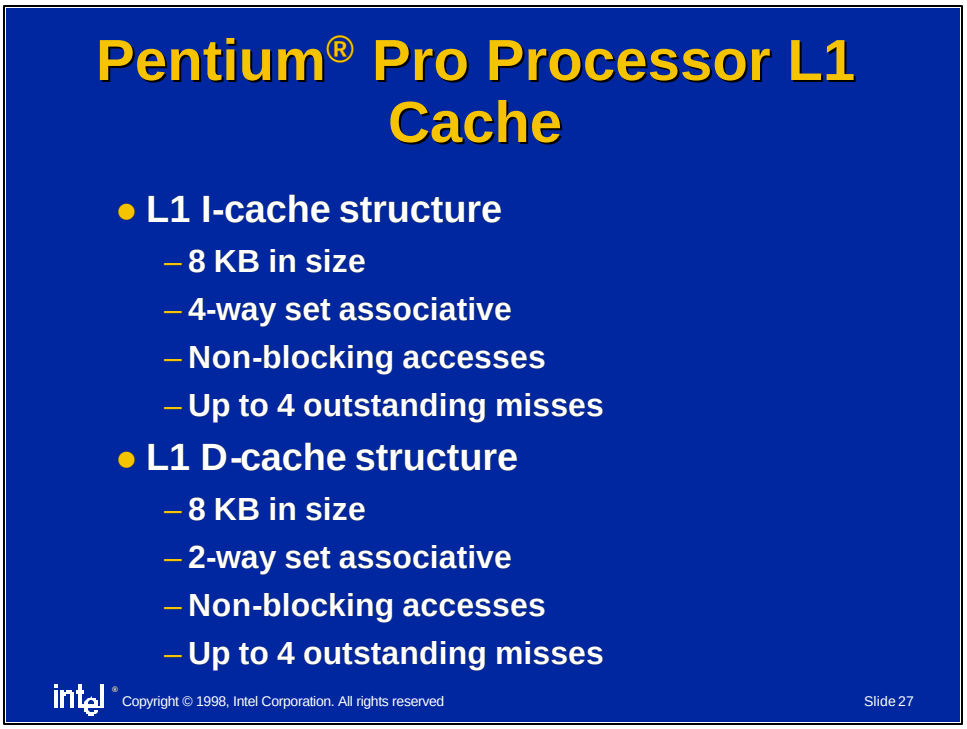


Notes

The following slides provides detailed information regarding cache memory within the P6 microarchitecture.

The above slide shows that the P6 microarchitecture CPU core includes a Level 1 (L1) Instruction cache and L1 Data cache. The L1 instruction cache is single ported while the L1 data cache is dual-ported. The Bus Interface Unit (BIU) is also integrated into the processor core. Circuits that interface the processor to the System Bus is included in the core as well.

A unified data and instruction Level 2 (L2) cache is integrated in the same package as the CPU core. The L2 cache is connected to the CPU core through separate bus - the L2 Cache Bus (or Backside Bus). Most P6 microarchitecture processors have L2 Cache Bus that runs at the same frequency as the CPU core.

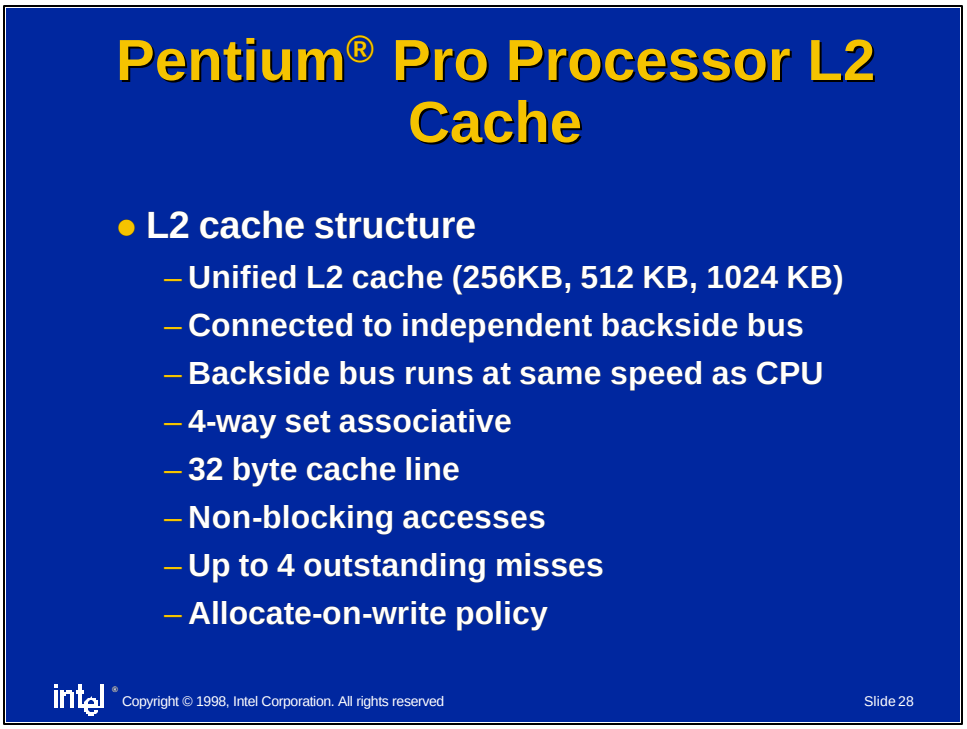


Notes

The sizes and configuration of the L1 caches on different P6 microarchitecture processors vary. However, each processor is configured so that the L1 instruction cache is separate from the L1 data cache.

The Pentium Pro processor has an L1 instruction cache that is a 4-way set associative 8KB cache. The L1 data cache is also 8KB in size. However, unlike the L1 instruction cache, the data cache is only 2-way set associative. Both caches support non-blocking accesses and can have up to 4 outstanding misses without stalling the processor.

The Pentium II processor has an L1 instruction cache and L1 data cache that are both 4-way set associative and 16KB in size. Both caches support non-blocking accesses and can have up to 4 outstanding misses without stalling the processor.

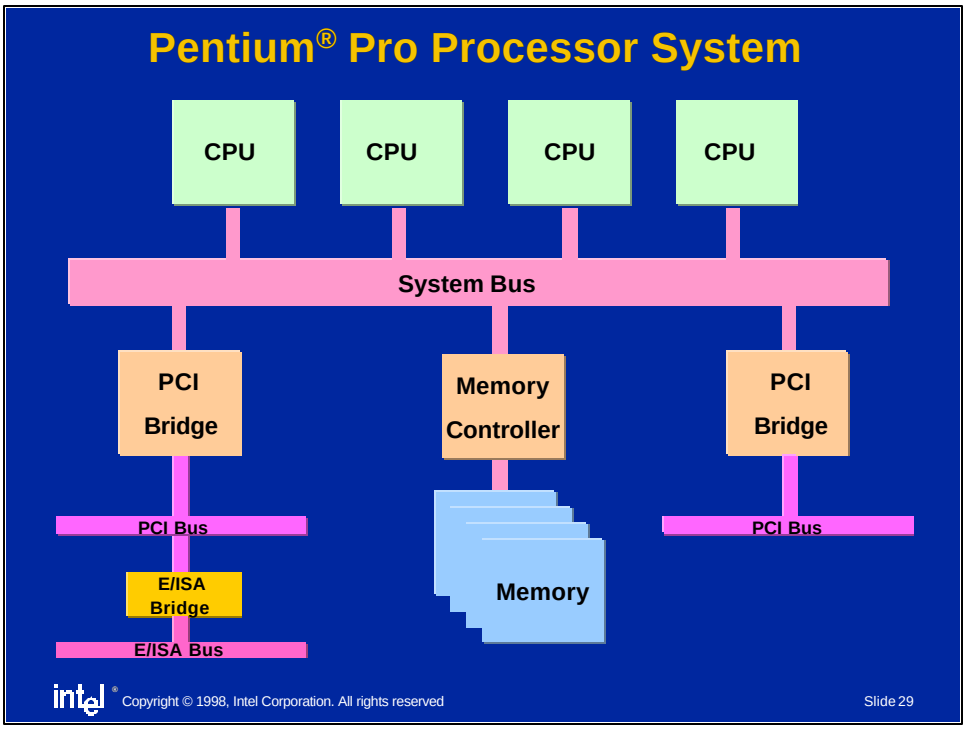


Notes

Processors based on the P6 microarchitecture all have a unified data and instruction L2 cache in the same package as the CPU. The L2 caches are all 4-way set associative caches. However, the L2 Cache Bus speed and sizes supported by each processor vary.

The Pentium Pro processor has an L2 Cache Bus running at the CPU core frequency. It supports 256KB, 512KB, or 1024KB L2 cache size configurations.

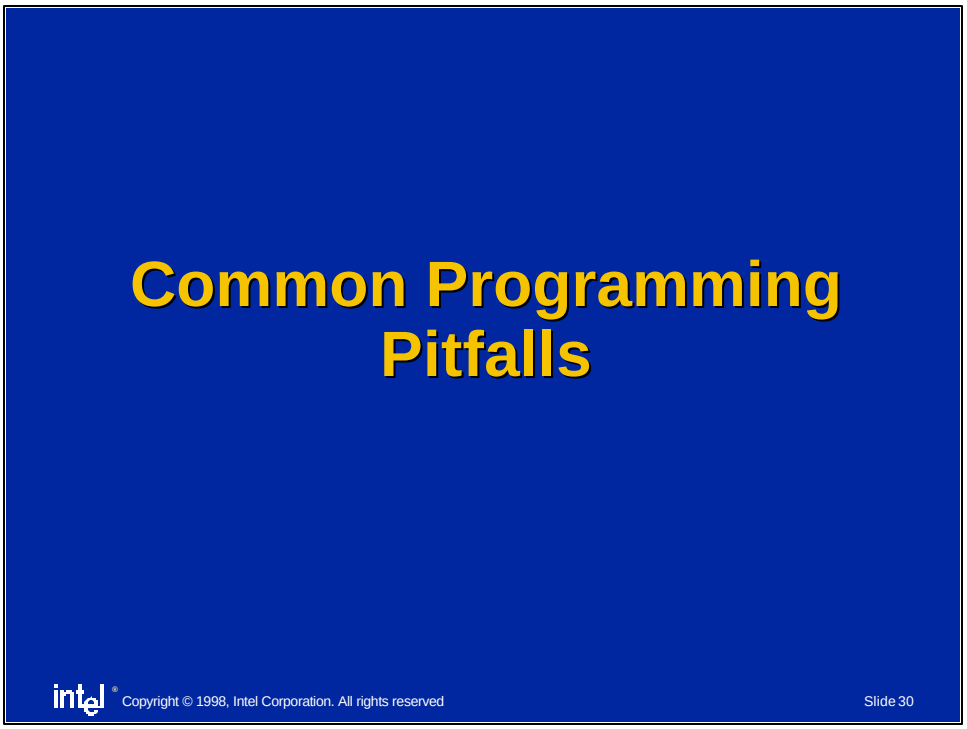
The Pentium II processor L2 Cache Bus runs at half the CPU core frequency. The Pentium II processor supports only 256KB and 512KB cache size configurations.



Notes

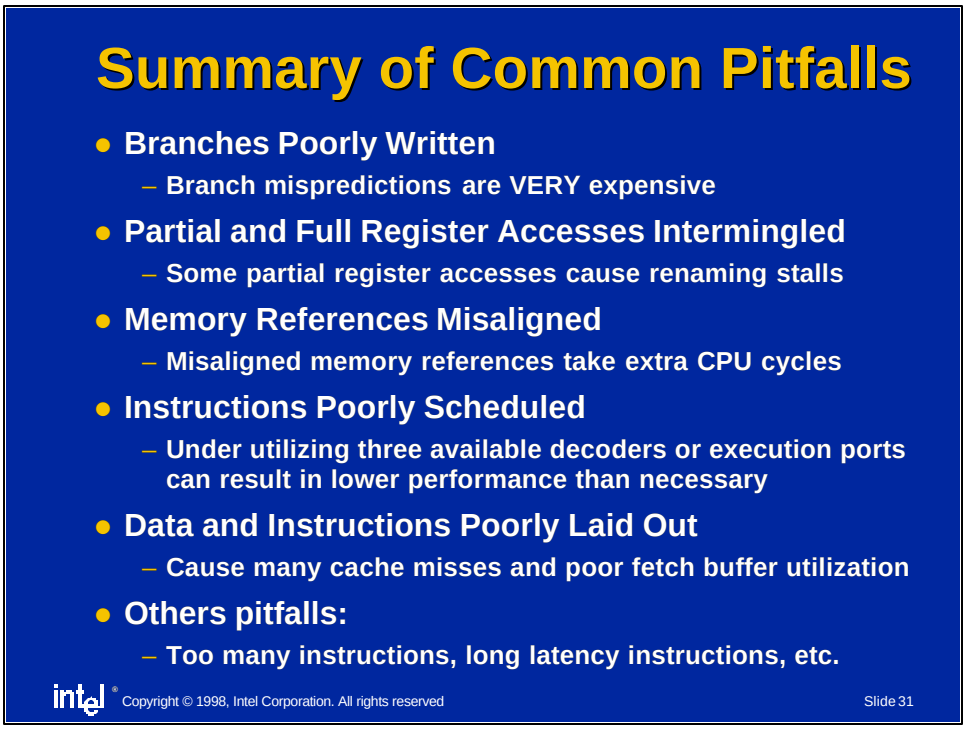
The figure above illustrate a typical design of a four processor Symmetric Multi-Processor (SMP) system based on the P6 microarchitecture. The system bus supports up to eight loads. Seven of the loads are taken up by processors and chipsets; one load is available for expansions such as clustering and additional memory controllers.

Four processors sit on the system bus as peers. Because of the integration of L2 cache into the CPU package, it is instructive to observe that L2 cache hits avoid bus transactions. Hence, performance of system based on the P6 microarchitecture can be significantly improved by improving the cache hit rate. Also, cache misses are also significant as they involve transaction on the relatively slower system bus.



Notes

The next set of slides summarizes the most common programming pitfalls for applications running on a P6 microarchitecture processor. Each pitfall is described in detail; solutions are also offered.



Notes

The above list summarizes some of the ways that an application performance on a P6 microarchitecture processor can be hindered. Application developers can use this list to systematically investigate how to optimize the processor performance for an application.

Many of the items in this list have corresponding events that are kept track of by the microarchitecture; some can be deduced from several other microarchitecture events.

The most common application issues include:

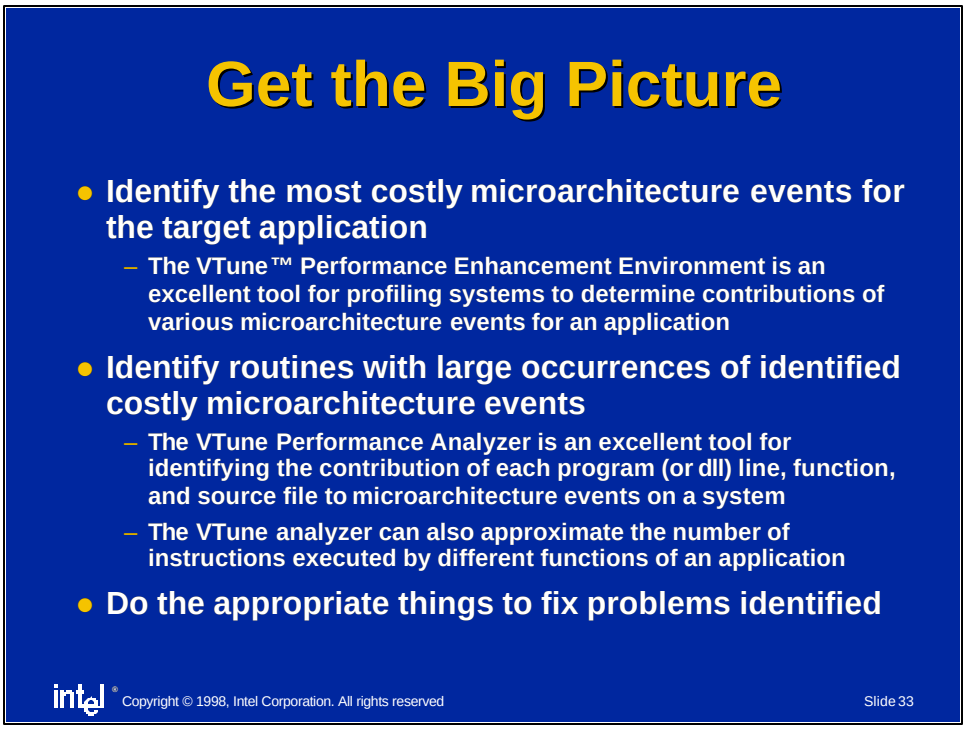
- Branch mispredictions which can occur if branches are written poorly
- Misaligned Memory References which occur when instructions and data are incorrectly organized in memory
- Decode stalls which can occur when the three available decoders are not fully utilized
- L2 Cache misses which can occur because of poor design of an application
- Poor execution throughput which can occur when the throughput of the front end microarchitecture is poor or when a lot of instructions need to use the same execution port (e.g. series of FDIV operations)

There are also other issues such as path-length (i.e. too many instructions executed per transaction) which may be detected at the microarchitecture tuning level but fixed at the system and application tuning levels.

Notes

Each pitfall and solutions are described in the next set of slides.

This section begins with a description of a generic methodology for resolving problems with the performance of applications at the microarchitecture level. Then, each application pitfall is described in details; solutions to the pitfalls are provided after each pitfall.



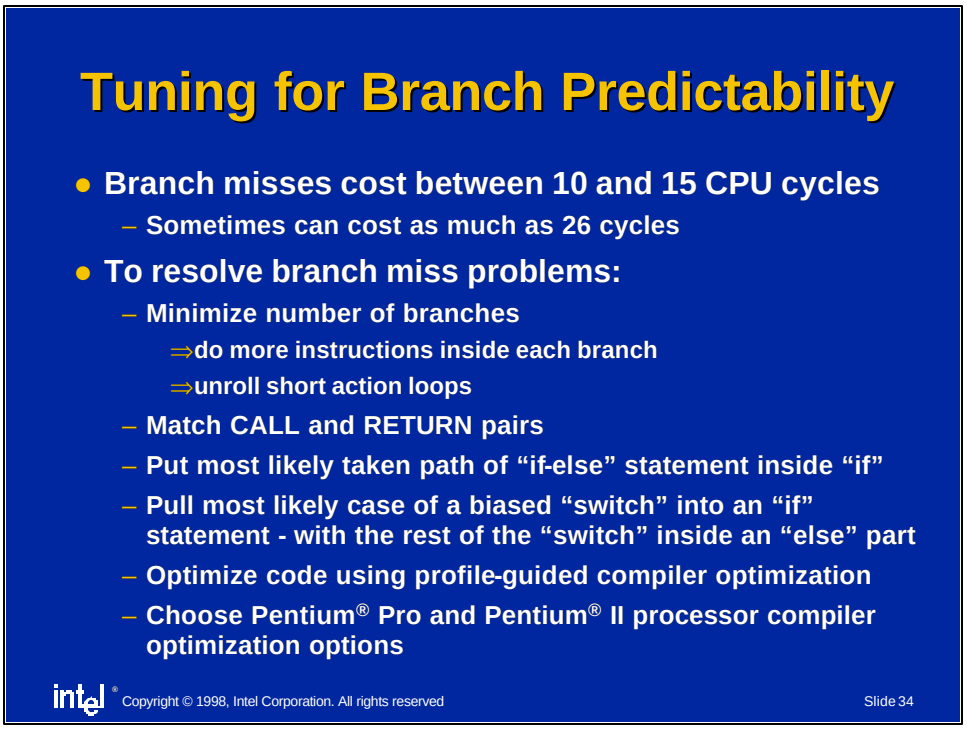
Notes

The following outline describes the steps that should be taken to identify and resolve microarchitecture performance issues for an application.

It is necessary to go after the biggest opportunity for performance improvement. Amdahl's Law can be applied to select which processor events are the most costly events for an application.

At the start, use the VTune analyzer to summarize the cost of each event in the list of microarchitecture events mentioned earlier in this presentation. Using the VTune Performance Analyzer summary costs of the events, choose the most costly events to optimize first; optimize the rest as time permits.

After identifying a costly event, use the VTune Performance Analyzer to identify the application location with the largest occurrence for the identified event. Using the Intel Architecture Optimization Manual and this presentation as guides, improve those portions of the application to eliminate the occurrences of the costly event. Follow the ASC process.



Notes

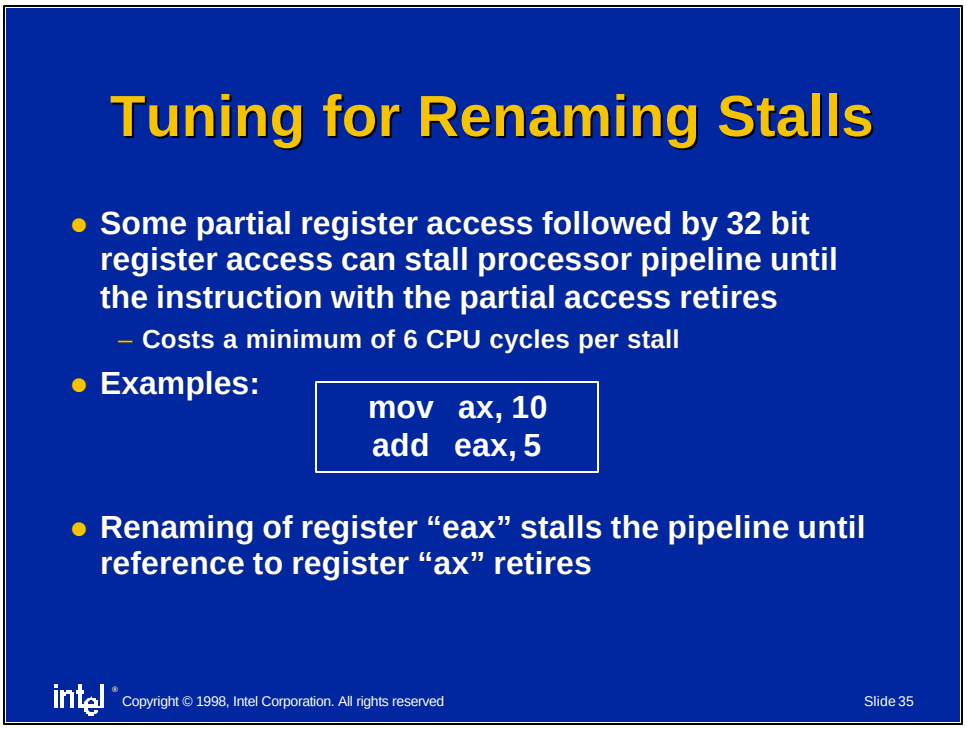
Instructions speculatively executed must be flushed from the processor pipeline after each branch misprediction is detected. This can result in a lot of wasted CPU cycles as new instructions need to be fetched into the pipeline. On the P6 microarchitecture, branch mispredictions cost about 10 to 15 CPU cycles; it can be as much as 26 cycles sometimes.

Branch misspredictions can easily be the most costly event for an application. Many large server and workstation applications have a lot of active branches. Some applications generate an average of one branch instruction for every 3 instructions. Therefore, there is a high probability that some poorly written branches can easily cause visible performance problems - as the mispredicted branches are restarted in the pipeline.

To minimize the probability of branches misprediction, it is necessary to reduce the number of branches in a program by doing more instruction inside each loop and unrolling short action loops. It is also important to write branches so that the static branch prediction rule is correct most of the time.

Most loop naturally work well under the static branch prediction rule. However, 'if-else' and 'switch' statements do not work well. However, a programmer can improve the performance of an 'if-else' statement by putting the most likely case of the statement inside the 'if' portion of the statement. The performance of 'switch' statements can be improved by pulling the most likely case of a highly biased (i.e. >90% of the time one case is taken) inside an 'if' statement.

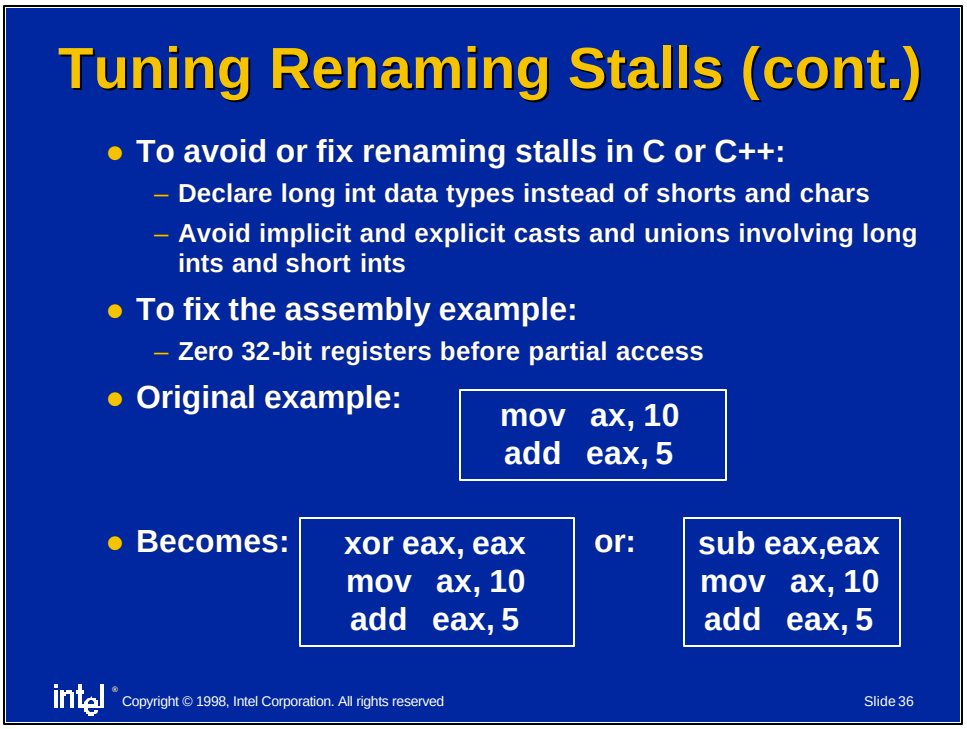
It is also important to use a P6 microarchitecture aware compiler so that the compiler can generate the right branch code for optimal performance of the application.



Notes

Renaming stalls usually occur if an application accesses a full (i.e. 32-bit register) soon after the application accesses the same register as a partial register. The example on this page illustrates this issue with two forms of access to the EAX register; register EAX is accessed both as a 16bit quantity - AX, and as a full 32bit quantity - EAX.

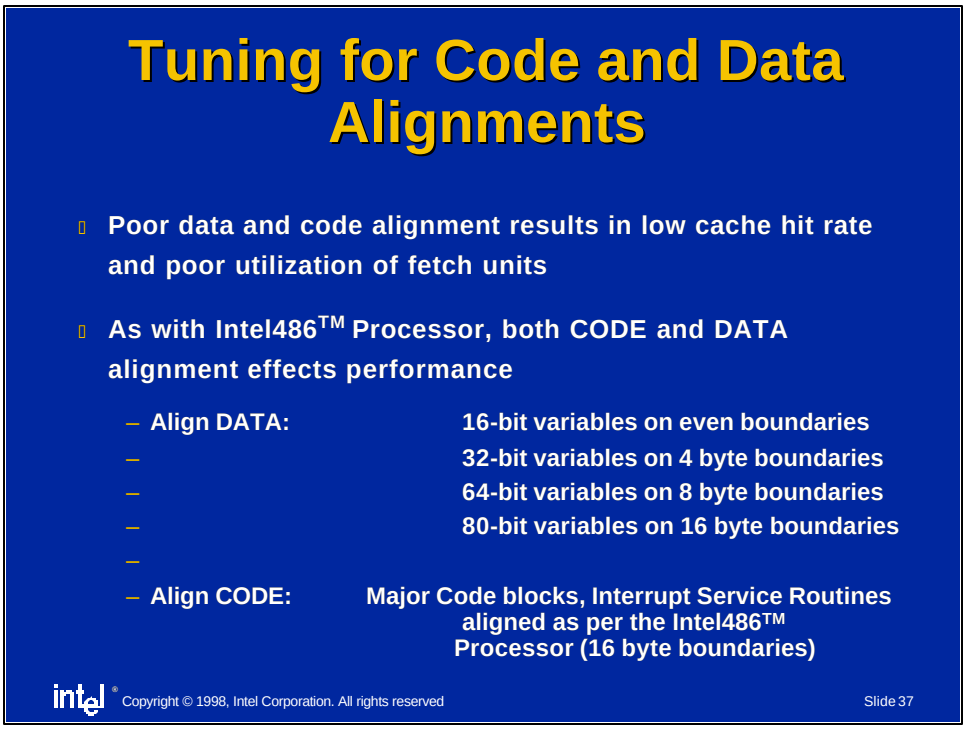
During these kinds of register accesses, the P6 microarchitecture processors stalls the processor pipeline until the instruction with the partial register access is retired before any proceeding instructions can progress through the processor pipeline. These stalls take a minimum of 6 CPU cycles. The number of cycles stalled is determined by the number of instructions pending in the ROB for execution and retirement before the instruction making the partial register access can be retired.



Notes

There are simple ways of removing the renaming stalls in C/C++ and in assembly languages. The best way to avoid such partial access in C and C++ is to avoid implicit or explicit casts between variables of different sizes.

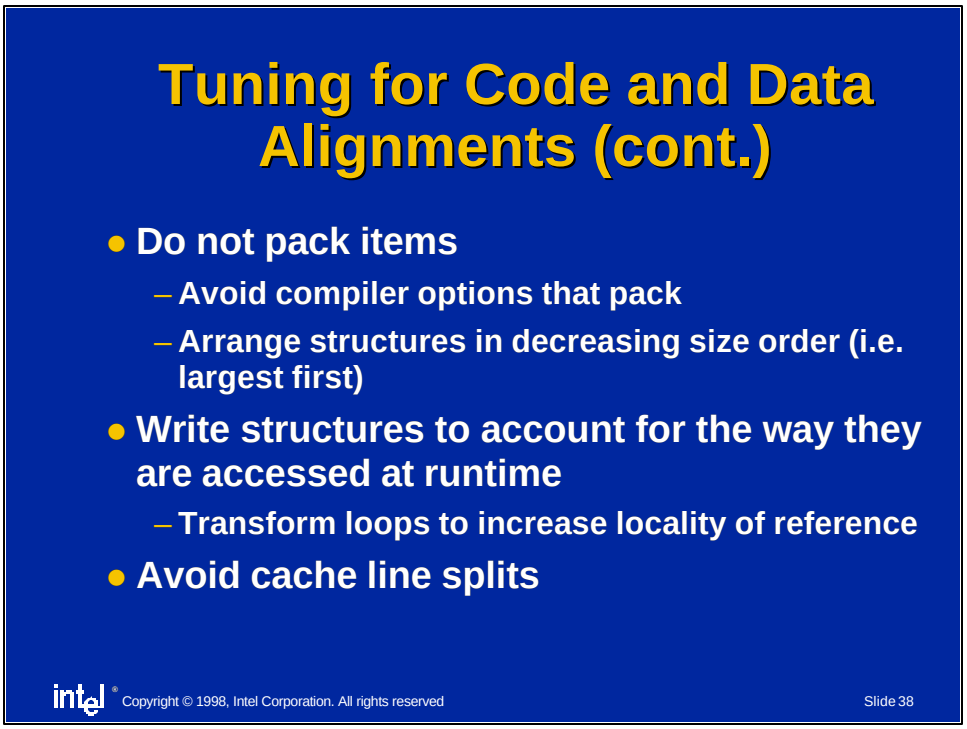
Unions involving data elements of differing sizes can also generate renaming stalls. Avoiding interleaved accesses to data elements of such unions will help to minimize renaming stalls.



Notes

The alignment of code and data in memory will also impact the processor performance of applications. Misaligned data and code take extra CPU cycles to fetch. Therefore, it is critical to have code and data elements aligned on their natural boundaries in memory. For example, 16-bit data should be aligned on even byte boundaries while 32-bit data should be aligned on 4-byte boundaries.

Since the Instruction Fetch Unit fetches 16 bytes of data at a time, code blocks - especially heavily accessed code such as loops and Interrupt Service Routines - should start at 16 byte boundaries.



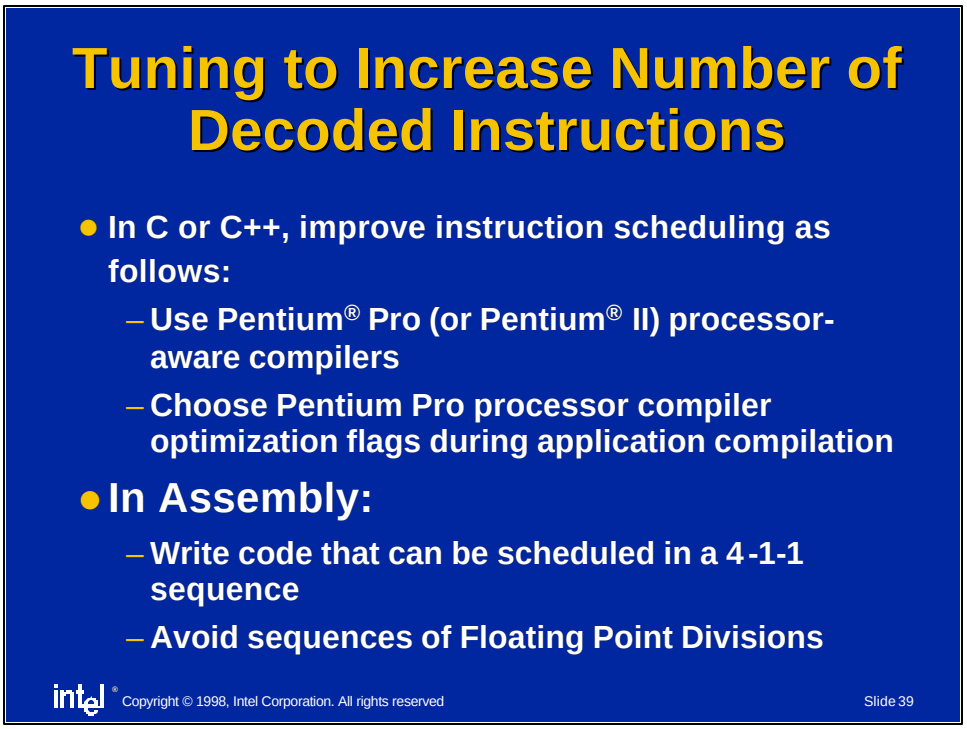
Notes

Application developers should pay special attention to the ordering of elements and structures in memory to avoid misaligned memory references.

To get good alignment of structures, the compiler ‘pack’ option should not be used during compilation. There are tricks that allow structures to take the least amount of memory as well as be properly aligned. By declaring the order of elements in a structure from the largest to the smallest, good alignment as well as packing can be achieved.

Also, structures should be written to increase spatial locality as well as referential locality at runtime. Structure elements that are referenced together should appear physically together in memory and vice versa.

It is also important to have structures appear in the minimum number of cache lines. Avoid having structures split between cache lines as accesses to these structures could potentially result in several cache misses.

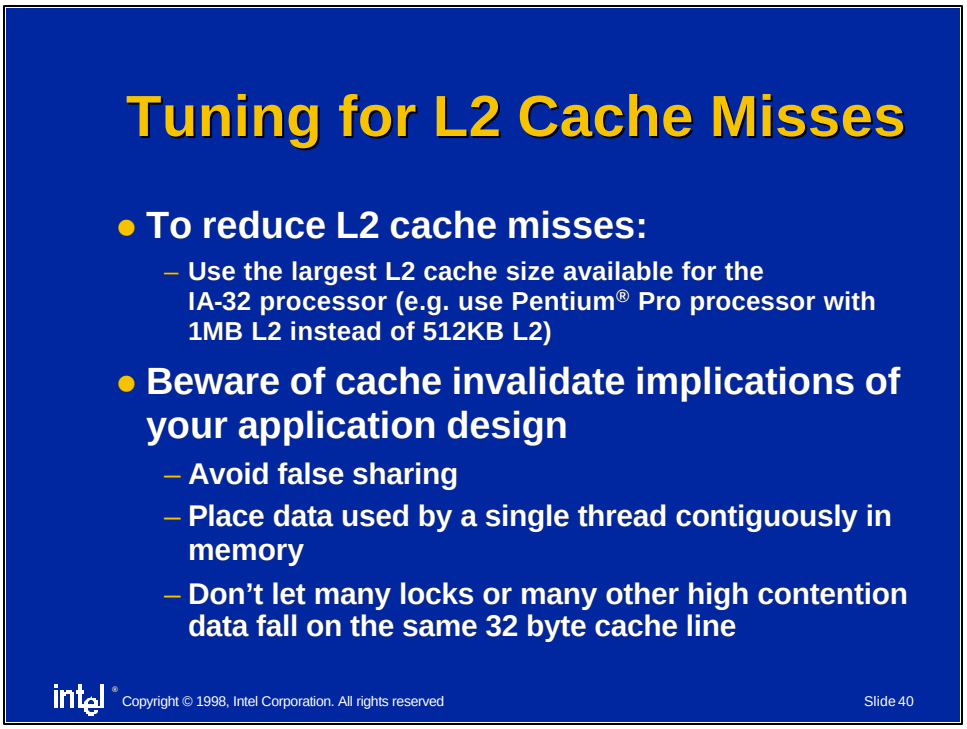


Notes

The configuration of three decoders in the P6 microarchitecture requires that the scheduling order of instructions is important.

For optimal performance of applications on the P6 microarchitecture, compilers and assemblers need to generate codes that appear in a 4-1-1 sequence. P6 microarchitecture aware compilers such as MS Visual Studio 5.0 or later and Intel Proton Compiler generate the right code sequence; older version of Microsoft compilers do not generate the right sequence.

The 4-1-1 sequence will result in good utilization of all available decoders. This, in turn, will help the throughput and performance of the front end microarchitecture.



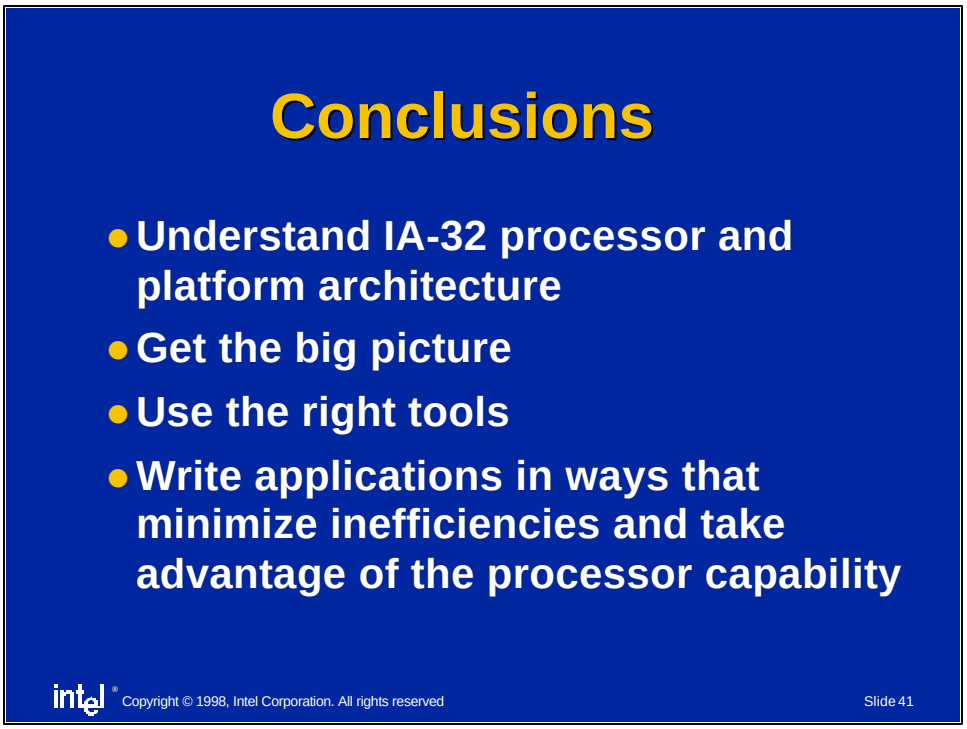
Notes

The Level 1 (L1) and Level 2 (L2) caches are used to minimize the impact of latency gap between accesses to CPU registers and accesses to main memory. As the gap between memory and CPU latencies widens, the importance of cache increases. As CPU frequencies increase, it is necessary that the most frequently used data and instructions are available in the fast (i.e. cache) memory. The likelihood of CPU pipeline stalls due memory requests missing the cache decreases with decreases in cache miss rate.

L2 cache miss rate indicates the ratio of all memory requests that were not satisfied by the L1 or the L2 cache. The organization of data and instructions in memory affects the L2 cache miss rate. Therefore, application developers should design code and data structures so that cache miss rates are minimized.

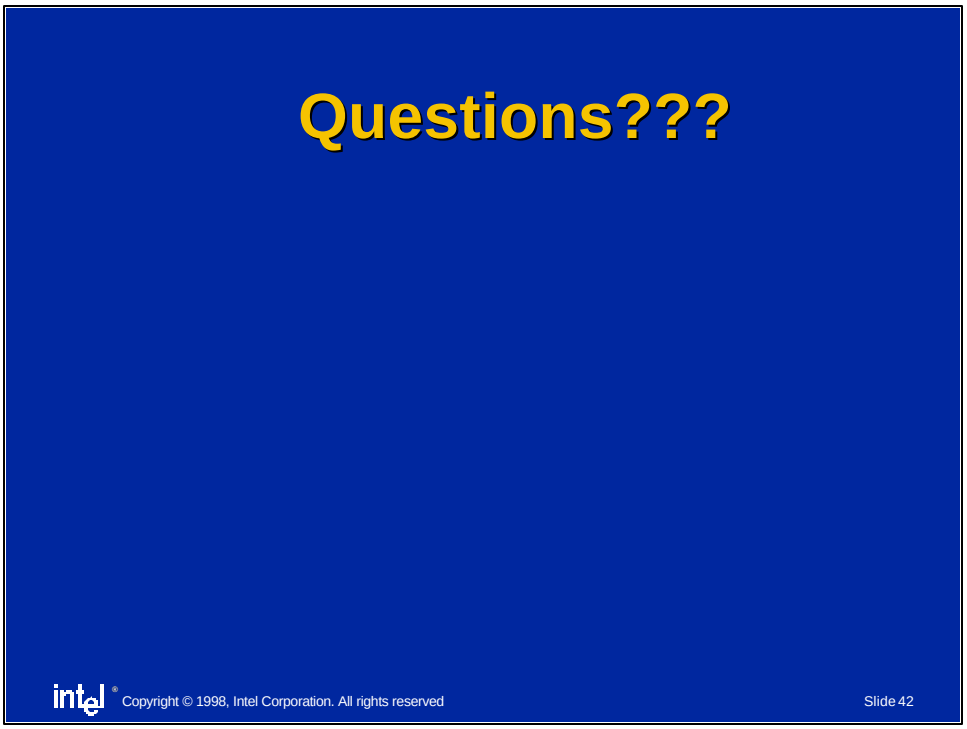
To minimize the L2 cache miss rate for an application that requires a lot of cache, use the IA-32 Architecture processor with the largest cache size configuration. Within an application, the organization of structures will impact the total number of cache misses. By implementing structures so that cache splits are avoided, programmers can also affect the overall cache miss rate for an application.

On SMP environment, cache misses may also occur on a processor when the cache line needed by the processor has been modified on the L2 cache of another processor. During these situations, cache line invalidate is initiated so that the processor with modified data can write the cache line back to memory so that the data is available to other processors. To avoid a lot cache line invalidates, data structures should be written so that false sharing of cache lines between processors executing different threads are avoided.

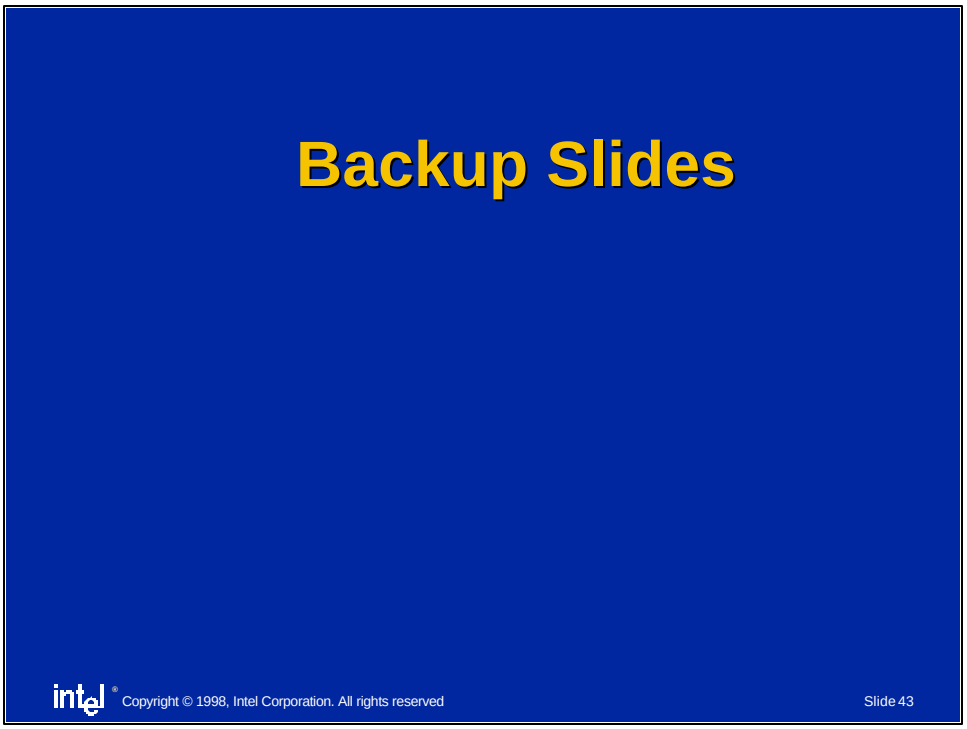


Notes

This presentation described the P6 microarchitecture and how to take advantage of its feature for optimal application performance. A general approach for using the build-in processor event counters to understand performance bottlenecks within the processor pipeline was also reviewed. The presentation concluded with a detailed description of the most common application programming pitfalls and how to eliminates them for optimal performance of applications.


Questions???
intel
Copyright © 1998, Intel Corporation. All rights reserved
Slide 42

Notes


Backup Slides
intel
Copyright © 1998, Intel Corporation. All rights reserved
Slide 43

Notes

Comparing L2 Cache Sizes

- Don’t compare IA-32 processor cache size with cache sizes of proprietary RISC processors
- RISC processors are typically based on fixed size instruction set (ISA)
  - Fixed size ISA processors have poor code density
- RISC processors also require 3 instructions for every CISC instruction on average
  - Thus, RISC processors have more code cache requirement
    - ⇒some RISC processors need 4 to 15 times larger cache size for similar performance as IA-32 processors for some applications
    - ⇒some RISC processors also need 2 to 5 times higher CPU frequency for comparable IA-32 processor performance

Slide 44

Notes